(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

IDENTIFICATION AND CONTROL OF NON-LINEAR TIME-VARYING DYNAMICAL SYSTEMS USING ARTIFICIAL NEURAL NETWORKS

by

Shahar Dror

September 1992

Dissertation Advisor: Daniel J. Collins

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704 0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable) AA	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

IDENTIFICATION AND CONTROL OF NON-LINEAR TIME-VARYING DYNAMICAL SYSTEMS USING ARTIFICIAL NEURAL NETWORKS

12 PERSONAL AUTHOR(S) Shahar Dror

13a TYPE OF REPORT Ph.D. Dissertation	13b TIME COVERED FROM ______ TO ______	14 DATE OF REPORT (Year, Month, Day) 1992, September	15 PAGE COUNT 285

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are of the author and not reflect the official policy or position of the Department of Defence or the U.S. Government.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Neural Networks, Identification, Adaptive Control, Non-Linear Systems, Time-Varying Dynamical Systems

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Identification and control of non-linear dynamical systems is a very complex task which requires new methods of approaching. This research addresses the problem of emulation and control via the use of distributed parallel processing, namely artificial neural networks. Four models for describing non-linear MIMO dynamical systems are presented. Based on these models a combined feedforward and recurrent neural networks are structured to emulate the dynamical system. Further, a procedure to emulate multiple systems in a single network is suggested. A method for finding a minimal realization of a network is introduced. The minimization greatly reduces the complexity of the network without degrading the operating performance of the network.

This work also examines the application of artificial neural networks for adaptive control. The multiple-system approach is used to find an adaptive neural network controller for non-linear MIMO time-varying system in a direct model reference control scheme. The controller network is trained using a procedure called back-propagation through the plant, which was extended in this work.

The application of neural networks is demonstrated on a longitudinal model of the F/A-18A fighter aircraft both with the undamaged aircraft and with a damage mechanism as a time-varying MIMO dynamical system.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Daniel J. Collins	22b TELEPHONE (Include Area Code) (408)646-2311	22c OFFICE SYMBOL AA/Co

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

Identification and Control of Non-Linear Time-Varying Dynamical Systems Using Artificial Neural Networks

by

Shahar Dror
Lieutenant Commander, Israeli Navy
B.Sc. A.E. Technion - Israel Institute of Technology, 1983
M.Sc. A.E. Technion - Israel Institute of Technology, 1989

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN AERONAUTICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Author: [illegible]
Shahar Dror

Approved by:

[illegible]
Brij N. Agrawal
Professor of Aeronautics

Daniel J Collins
Daniel J. Collins
Professor of Aeronautics
Dissertation Supervisor

Louis V. Schmidt
Louis V. Schmidt
Professor of Aeronautics

[illegible]
Arthur L. Schoenstadt
Professor of Mathematics

[illegible]
Harold A. Titus
Professor of Electrical Engineering

Approved by: Daniel J Collins
Daniel J. Collins, Chairman,
Department of Aeronautics and Astronautics

Approved by: [illegible]
Richard S. Elster, Dean of Instruction

ABSTRACT

Identification and control of non-linear dynamical systems is a very complex task which requires new methods of approaching. This research addresses the problem of emulation and control via the use of distributed parallel processing, namely artificial neural networks. Four models for describing non-linear MIMO dynamical systems are presented. Based on these models a combined feedforward and recurrent neural networks are structured to emulate the dynamical system. Further, a procedure to emulate multiple systems in a single network is suggested. A method for finding a minimal realization of a network is introduced. The minimization greatly reduces the complexity of the network without degrading the operating performance of the network.

This work also examines the application of artificial neural networks for adaptive control. The multiple-system approach is used to find an adaptive neural network controller for non-linear MIMO time-varying system in a direct model reference control scheme. The controller network is trained using a procedure called back-propagation through the plant, which was extended in this work.

The application of neural networks is demonstrated on a longitudinal model of the F/A-18A fighter aircraft both with the undamaged aircraft and with a damage mechanism as a time-varying MIMO dynamical system.

THESIS DISCLAIMER

TABLE OF CONTENTS

I. INTRODUCTION 1

A. RESEARCH OBJECTIVE 2

B. A BRIEF SURVEY OF NEURAL NETWORKS INVOLVING DYNAMICAL SYSTEMS 2

C. THE CONTRIBUTIONS OF THIS WORK 3

D. ORGANIZATION OF THE DISSERTATION 4

II. NEURAL NETWORK THEORY 6

A. INTRODUCTION TO NEURAL NETWORKS 6

B. ARTIFICIAL NEURAL NETWORK ARCHITECTURE 7

1. Environment 8

2. Propagation Scheme 9

3. Learning Rule 9

4. Recall Phase 10

C. ARTIFICIAL NEURAL NETWORK STRUCTURE 11

1. Multi-Layer Neural Network Structure 11

2. Processing Element 13

3. Activation Function of a Processing Element 15

D. CATEGORIES OF MULTI-LAYER NETWORKS 17

E. ARTIFICIAL NEURAL NETWORKS AS GENERAL APPROXIMATORS 18

III. ANALYSIS OF BACK-PROPAGATION ALGORITHM 21

A. ARCHITECTURE 21

B. MINIMIZING GLOBAL MEASURE 22

C. BACK-PROPAGATION LEARNING RULE DERIVATION 24

D. COMMON VARIATIONS ON THE STANDARD BACK-PROPAGATION ALGORITHM 29

1. Momentum Term 30

2. Cumulative Update of Weights 30

3. Fast Back-Propagation 30

4. Modifying the Derivative 31

E. BACK PROPAGATION THROUGH THE PLANT 31

1. Architecture and Formulation 32

2. Derivation of Learning Rule for Back-Propagation Through Plant 33

3. Estimating the Jacobian of the Plant 35

IV. NEURAL NETWORK REPRESENTATION OF DYNAMICAL SYSTEMS 38

A. CHARACTERIZATION OF NONLINEAR DYNAMICAL SYSTEMS 39

1. Model I 40

2. Model II 40

3. Model III 41

4. Model IV 41

B. NEURAL NETWORK REPRESENTATION OF NON-LINEAR DYNAMICAL SYSTEMS 42

1. Structural Aspects 45

2. Trainability Aspects 48

C. ARTIFICIAL NEURAL NETWORK REPRESENTATION OF LINEAR TIME-INVARIANT SYSTEMS 50

D. EMULATION OF MULTIPLE MIMO SYSTEMS IN A SINGLE ANN 51

V. ARTIFICIAL NEURAL NETWORKS IN ADAPTIVE CONTROL 53

A. BASIC CONCEPTS OF ADAPTIVE CONTROL 53

1. Model-Reference Adaptive Control 55

B. ADAPTIVE CONTROL USING ARTIFICIAL NEURAL NETWORKS 57
1. Imitating Classical Methods 57
2. Inverse-dynamics Control 58
3. Model-Reference Control Using Neural Networks 60
C. ANN ADAPTIVE CONTROLLER FOR TIME VARYING DYNAMICAL SYSTEMS 63
1. Design Philosophy 64
2. ANN Structure for the Unified Controller 65

VI. DESIGN CONSIDERATIONS 68
A. DYNAMICAL SYSTEM PROPERTIES 69
1. System Properties for Emulation Design 69
2. System Properties for Controller Design 69
B. SPECTRUM AND AMPLITUDE ISSUES 69
C. NETWORK STRUCTURE SELECTION 70
D. MINIMAL REALIZATION OF A MULTI-LAYER ANN 73
1. Selection of Number of PEs in Hidden Layers 75
2. Selection of Number of Hidden Layers 78
3. Selection of Regression Order 78
E. VALIDATION ISSUES 79

VII. ANN EMULATION AND CONTROL DEMONSTRATION . 81

A. F/A-18A LONGITUDINAL DYNAMICAL MODEL 81

B. DAMAGED F/A-18A MODEL 84

C. CONTROL REFERENCE MODEL 85

D. DESIGN PARAMETERS AND CONDITIONS 86

1. Plant Properties 86

2. Spectrum and Amplitude Ranges 86

3. Network Structure for Emulation and Control of the F/A-18 Model 89

VIII. SIMULATIONS, RESULTS AND ANALYSIS 90

A. EMULATION OF A SINGLE MIMO DYNAMICAL SYSTEM . . . 91

1. Single Plant Emulation - Frequency Domain Results and discussion 93

2. Single Plant Emulation - Time Domain Results and Discussion . 99

B. EMULATION OF MULTIPLE PLANTS IN A SINGLE NETWORK 104

1. Multiple-Plant Emulation - Frequency Domain Results and discussion 104

2. Multiple-Plant Emulation - Time Domain Results and Discussion 108

C. MINIMAL REALIZATION OF A SINGLE-NETWORK MULTIPLE-PLANT EMULATOR 114

1. Minimal Realization of a Multiple-Plant Network - Frequency Domain Results and Discussion 118

2. Minimal Realization of a Multiple-Plant Network - Time Domain Results and Discussion 122

D. EMULATION ANALYSIS 128

E. ADAPTIVE CONTROL OF MIMO DYNAMICAL SYSTEM 129

1. Model-Reference Controller - Frequency Domain Results and Discussion 129

2. Model-Reference Controller - Time Domain Results and Discussion 133

3. Control Analysis 134

IX. CONCLUSIONS AND RECOMMENDATIONS 141

A. MAIN RESULTS 141

B. RECOMMENDATIONS FOR FURTHER RESEARCH 142

APPENDIX A. F/A-18A MATHEMATICAL MODEL 144

APPENDIX B. TIME AND FREQUENCY RESPONSES OF ARTIFICIAL NEURAL NETWORK EMULATION OF DAMAGED AND UNDAMAGED F/A-18A AIRCRAFT . . . 149

APPENDIX C. SINGULAR VALUES PLOTS 214

APPENDIX D. USERIO EMULATION PROGRAMS 218

LIST OF REFERENCES 252

INITIAL DISTRIBUTION LIST 257

LIST OF FIGURES

Figure 2.1: A Basic Schematic of an Artificial Neural Network 7

Figure 2.2: A Three-Layer Feed-Forward Artificial Neural Network 12

Figure 2.3: A Basic Configuration of a Processing Element 14

Figure 2.4: Hyperbolic Tangent and Sigmoid Activation Functions and Derivatives 16

Figure 2.5: A Recurrent Network 17

Figure 2.6: General Multi-Layer Building Block 17

Figure 2.7: General Regression Network 17

Figure 3.1: A Diagrammatic Representation of Back-Propagation Architecture 28

Figure 3.2: Artificial Neural Network preceding a Physical Process 32

Figure 4.1: A Neural Network Emulation of Dynamical System 44

Figure 4.2: Artificial Neural Network Emulator for Models I-III 47

Figure 4.3: A Neural Network Representation of LTI System 51

Figure 5.1: Direct Adaptive Control 56

Figure 5.2: Indirect Adaptive control 56

Figure 5.3: Direct Inverse Control 59

Figure 5.4: Model-Reference Adaptive Control Architecture 60

Figure 5.5: Indirect Model-Reference Adaptive Control 62

Figure 5.6: Direct Model-Reference Adaptive Control 63

Figure 5.7: Control Architecture 64

Figure 5.8: Unified Controller Training via BPTP in a Model-Reference Scheme66

Figure 5.9: General Structure of ANN Unified Controller 66

Figure 6.1: A General Structure for Multi-Layer ANN 72

Figure 6.2: Singular Values of a Weight Matrix as a Function of Training Iteration 77

Figure 6.3: A Weight Matrix From Regression Layer to Hidden Layer 78

Figure 7.1: F/A-18A Axes, Control Surfaces and Sign Conventions 82

Figure 7.2: Model-Reference Control Architecture for the F/A-18A 85

Figure 7.3: Time Domain Sample of Random Binary Sequence 88

Figure 7.4: Power Spectrum of Training RBS Signal (512k presentations) . . . 88

Figure 7.5: Power Spectrum of Test RBS Signal (16k presentations) 88

Figure 8.1: Network Structure for a Multiple-Plant F/A-18 Emulation 92

Figure 8.2: Spectral Response of Undamaged Aircraft Due to Stabilators . . . 97

Figure 8.3: Spectral Response of Damaged Aircraft Due to Stabilators 98

Figure 8.4: Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Command in the Stabilators 102

Figure 8.5: Normal Acceleration Response of the Damaged Aircraft Due to Dipole Command in the Stabilators 103

Figure 8.6: Spectral Response of Undamaged Aircraft Due to Stabilators in a Multiple-Plant Single Network Emulation 106

Figure 8.7: **Spectral Response of Damaged Aircraft Due to Stabilators in a Multiple-Plant Single Network Emulation** 107

Figure 8.8: **Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Command in the Stabilators in a Multiple-Plant Network** 110

Figure 8.9: **Normal Acceleration Response of the Damaged Aircraft Due to Dipole Command in the Stabilators in a Multiple-Plant Network** 111

Figure 8.10: **Normal Acceleration Response of the Single Network Emulating Two Plants Due to RBS Excitation of All Inputs** 113

Figure 8.11: **Singular Values Plot of the Non-Minimized Network** 116

Figure 8.12: **Singular Values Plot of the Minimized Network** 117

Figure 8.13: **Minimal Realization of a Network for Multiple-Plant F/A-18 Emulation** 117

Figure 8.14: **Spectral Response of Undamaged Aircraft Due to Stabilators in a Minimally Realized Multiple-Plant Single Network Emulator** 120

Figure 8.15: **Spectral Response of Damaged Aircraft Due to Stabilators in a Minimally Realized Multiple-Plant Single Network Emulator** 121

Figure 8.16: **Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Stabilators Command in a Minimized Multiple-Plant Network** 124

Figure 8.17: **Normal Acceleration Response of the Damaged Aircraft Due to Dipole Stabilators Command in a Minimized Multiple-Plant Network** 125

Figure 8.18: **Normal Acceleration Response of a Minimally Realized Single Network Emulating Two Plants Due to RBS Excitation of All Inputs** 127

Figure 8.19: **Spectral Response of Undamaged and Damaged Controlled Aircraft Due to Pilot Longitudinal Stick Command, Single Controller** 131

Figure 8.20: **Spectral Response of Undamaged and Damaged Controlled Aircraft Due to Pilot Longitudinal Stick Command, Unified Controller** 132

Figure 8.21: **Time Response of Undamaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller** 135

Figure 8.22: **Time Response of Damaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller** 136

Figure 8.23: **Actuation Commands of Undamaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller** 137

Figure 8.24: **Actuation Commands of Damaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller 138**

Figure 8.25: **Time response of Controlled Time Varying Aircraft Due to Pilot RBS Stick Excitation, Unified Controller 139**

Figure 8.26: **Actuation Commands of Controlled Time Varying Aircraft Due to Pilot RBS Stick Excitation, Unified Controller 140**

Figure B.1: **Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Single Plant, Single Network Emulation 150**

Figure B.2: **Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Single Plant, Single Network Emulation 151**

Figure B.3: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Single Plant, Single Network Emulation 152**

Figure B.4: **Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Single Plant, Single Network Emulation 153**

Figure B.5: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation . . 154**

Figure B.6: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation 155**

Figure B.7: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation 156**

Figure B.8: **Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 157

Figure B.9: **Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 158

Figure B.10: **Angle of Attack Response of Undamaged Aircraft Due to Stabilators, Dipole Command, Single Plant, Single Network Emulation** 159

Figure B.11: **Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 160

Figure B.12: **Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 161

Figure B.13: **Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation** . . 162

Figure B.14: **Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 163

Figure B.15: **Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 164

Figure B.16: **Normal Acceleration Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 165

Figure B.17: **Normal Acceleration Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 166

Figure B.18: **Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation** 167

Figure B.19: **Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 168

Figure B.20: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation** 169

Figure B.21: **Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Single Network Emulation** 170

Figure B.22: **Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Single Network Emulation** 171

Figure B.23: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Single Network Emulation** 172

Figure B.24: **Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Single Network Emulation** 173

Figure B.25: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation** 174

Figure B.26: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 175

Figure B.27: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 176

Figure B.28: **Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 177

Figure B.29: **Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 178

Figure B.30: **Angle of Attack Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation** 179

Figure B.31: **Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 180

Figure B.32: **Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 181

Figure B.33: **Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation** 182

Figure B.34: **Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 183

Figure B.35: **Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 184

Figure B.36: **Normal Acceleration Response Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 185

Figure B.37: **Normal Acceleration Response Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 186

Figure B.38: **Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation** 187

Figure B.39: **Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 188

Figure B.40: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation** 189

Figure B.41: **Pitch Rate Response to RBS Excitation of All Inputs, Multiple-Plant, Single Network Emulation** 190

Figure B.42: **Angle of Attack Response to RBS Excitation of All Inputs, Multiple-Plant, Single Network Emulation** 191

Figure B.43: **Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Minimized Network Emulation** 192

Figure B.44: **Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Minimized Network Emulation** 193

Figure B.45: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Minimized Network Emulation** 194

Figure B.46: **Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Minimized Network Emulation** 195

Figure B.47: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation** 196

Figure B.48: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 197

Figure B.49: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 198

Figure B.50: **Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 199

Figure B.51: **Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 200

Figure B.52: **Angle of Attack Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation** 201

Figure B.53: **Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 202

Figure B.54: Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation 203

Figure B.55: Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation 204

Figure B.56: Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation 205

Figure B.57: Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation 206

Figure B.58: Normal Acceleration Response Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation 207

Figure B.59: Normal Acceleration Response Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation 208

Figure B.60: Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation 209

Figure B.61: **Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 210

Figure B.62: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation** 211

Figure B.63: **Pitch Rate Response to RBS Excitation of All Inputs, Multiple-Plant, Minimized Network Emulation** 212

Figure B.64: **Angle of Attack Response to RBS Excitation of All Inputs, Multiple-Plant, Minimized Network Emulation** 213

Figure C.1: **Singular Values Plot of Weight Matrix from *ComReg* to *comh1*, Before Minimization** 215

Figure C.2: **Singular Values Plot of Weight Matrix from *ComReg* to *comh1*, After Minimization** 215

Figure C.3: **Singular Values Plot of Weight Matrix from *comh1* to *comh2*, Before Minimization** 216

Figure C.4: **Singular Values Plot of Weight Matrix from *comh1* to *comh2*, After Minimization** 216

Figure C.5: **Singular Values Plot of Weight Matrix from *OutReg* to *outh1*, Before Minimization** 217

Figure C.6: **Singular Values Plot of Weight Matrix from *OutReg* to *outh1*, After Minimization** 217

ACKNOWLEDGMENTS

As I sit down to conclude this research with a great sense of satisfaction and relief, it is difficult to find words to adequately express my appreciation for the contribution of many people who have made the completion of this work possible.

Special thanks go to my advisor, Professor Dan Collins, for his advice and guidance throughout the course of this research. During the many hours of formal and informal meetings and discussions, his valuable comments and suggestions helped me to successfully complete this thesis.

I would also wish to thank Professor Louis Schmidt, who served in my doctoral committee, for his support and for his thorough review and constructive comments to this manuscript. Gratitude is extended to Professors B. Agrawal, A. Schoenstadt and H. Titus, that as members of my doctoral committee lent their knowledge and experience.

Next, I wish to express my gratitude to my friends and fellow students for their friendship and encouragement, which made the Ph.D. experience easier and more enjoyable. In particular I would like to thank Dr. Moshe Idan, Denis Bertrand, Tony Cricelli, Jim Clifton and many others.

I would also like to thank the Israeli Navy for providing me with the opportunity to learn at the Naval Postgraduate School.

Finally, I wish to thank my wife Idit and my daughter Amit who have had to sacrifice greatly to allow me to get to this point. Without their love and support this dissertation would not have been possible.

לרעייתי עידית ולבתי עמית

To my beloved wife Idit who makes everything enjoyable, and to my precious daughter Amit who makes everything worthwhile.

I. INTRODUCTION

In many engineering and scientific applications a dynamical system or process of unknown structure acts upon measurable input excitations to produce observable output signals. One way of gaining knowledge of the system is by analyzing available data using a chosen candidate model to represent the system and applying an algorithm to adaptively change the parameters in the model, to cause it to function similarly to the unknown system when driven by the same inputs. The procedure is called *system identification* [LL87]. Controlling unknown systems is approached mainly by *robust control* and *adaptive control,* both relying on linearized or linear system theory to compensate for changes in the system. While robust control stays in the linear regime based on state feedback, adaptive control makes use of identification techniques to find a controller to compensate for parameter variations in the system as well as in the selected model, and regulates the output of the system according to some linear reference, resulting in a non-linear controller [AW89].

This work explores the subjects of identification and control of non-linear dynamical systems from aspects of artificial neural networks. This chapter describes the research objectives and contains a survey of previous work in the area of utilization of neural networks regarding dynamical systems. This chapter also summarizes the main contributions of the research and concludes with the organization of the dissertation.

A. RESEARCH OBJECTIVE

The research focuses on utilizing artificial neural networks with non-linear time-varying multi-input multi-output (MIMO) dynamical systems. The objective is to emulate multiple dynamical systems in a single neural network and to apply the procedure to form an adaptive controller for a changing plant.

The F/A-18 fighter aircraft was chosen to represent a MIMO dynamical system. The undamaged and impaired aircraft represented multiple models, for which an adaptive model-reference neural controller was designed.

B. A BRIEF SURVEY OF NEURAL NETWORKS INVOLVING DYNAMICAL SYSTEMS

Recently, there has been tremendous interest in exploiting artificial neural networks (ANN) to perform tasks of system identification and control [MSW90], [SM92]. The identification is done by *emulation* of the dynamical system *without* having to find specific parameter values of a hopefully matching model. Techniques from linear and adaptive control were used to select a proper network structure for the task, as was done by Narendra and Parthasarathy for emulation of nonlinear systems [NP90], by Scott and Collins [SC90] and Bertrand and Collins [BC92] who used aircraft models for emulation and control of linear systems, and by Barto, Sutton and Anderson for control of nonlinear plant of inverted pendulum [BSA83]. Controlling took shape in different ways, usually with single-input systems as like Guez, Eilbert and Kam did for second order linear system [GEK88], or as Ha, Wei and Bessolo did for linearized second order approximation of the F16 aircraft. The works of Kraft and Campagna, who used a memory addressable type neural network controller [KC90], and of Nguyen and Widrow for the truck backer-upper using sequential emulation and control learning [NW90] in nonlinear systems, are other examples of single-input systems.

For emulation and control the general approach has been to have a full state output, as Goldenthal and Farrell [GF90] and Anderson [AC89] did for inverted pendulum, as did Specht in his general regression neural network [SD91], and as was done in most of the works mentioned earlier. The approaches to control varied from neural network gain scheduling of classical controller of an aircraft by Burgin and Schnetzler [BS90], to finding an inverse plant such as was done by Bertrand [BD91] and by Levin, Gewirtzman and Inbar in a binary type inverse controller [LGI91], to self tuning or model reference adaptive control as was done in [KC90] and by Saerens, Soquet, Renders and Bersini using back-propagation through the plant training [SSRB90].

The field of multiple multi-input multi-output systems requires further research and exploration. It is the purpose, subject and substance of this work to expand the knowledge and understanding of such systems.

C. THE CONTRIBUTIONS OF THIS WORK

The research defines an approach to emulating non-linear MIMO dynamical systems using artificial neural networks, as well as the limitations associated with the systems and networks. Furthermore, the concepts developed for emulation are used to find a model-follower neural network controller for a MIMO system.

The research addresses the issue of treating multiple models in a single neural network associated with emulation and adaptive control of general dynamical systems from a common class. A general scheme is presented for training a single network with multiple processes sharing a common dynamical structure and the same order of input and output. Additionally,a method for finding minimal realization of an artificial neural network, using singular value decomposition of the weight matrices, is developed.

D. ORGANIZATION OF THE DISSERTATION

Dynamical systems in association with neural networks are addressed in several chapters of this work. The disclosure of what a neural network is and how it is trained, is described in chapters II and III. The architecture, structure and behavior of artificial neural networks are presented in chapter II. The back-propagation algorithm used in this work to train the networks is presented with a few variations and extensions in chapter III. Chapter IV characterizes the classes of dynamical systems with the associated structural aspects and training of neural networks used to emulate the systems. Chapter IV also present the approach to handling multiple models in a single network. A brief overview on adaptive control with implementation of control using neural networks is given in chapter V, which also establishes the procedure of reconfigurable neural-net controller. After having an understanding of what artificial neural networks are, how they are trained and how they specifically apply to the problems of emulation and control of MIMO non-linear dynamical systems, some design considerations for dealing with dynamical systems are highlighted in chapter VI. Chapter VII describes the F/A-18 aircraft as the dynamical system chosen to demonstrate the procedures which were developed in the previous chapters and is followed by a presentation and a discussion on the results in chapter VIII. Chapter IX closes with conclusions and recommendations for further research. To summarize, the research shows how identification and adaptive control problems can be solved quite straightforward using neural networks.

II. NEURAL NETWORK THEORY

A. INTRODUCTION TO NEURAL NETWORKS

A neural network, or better called *artificial neural network*, is a parallel, distributed information processing structure consisting of *processing elements* that are interconnected among themselves via unidirectional *connections*. The artificial neural network (ANN) has the ability to learn from experience, to perform abstractions of inputs with relevant information, and to generalize its knowledge from previous results. The massively connected network is highly parallel because the operations of each processing element occur simultaneously. Associated with each connection is a *weight*. The connection weights are the dynamic elements that change when the network is trained. The goal of the training is to adapt the weights so that the network performs in a desired fashion in response to a given input.

The architecture of artificial neural-network has three main levels: topological, data flow, and neurodynamics. The architectural and algorithmic factors that compose any neural network are: the interface with the environment, the propagation paradigm, the learning rule and the recall or test phase. The structure and behavior of artificial neural networks are set by the layers that are built of processing elements and the activation function of each processing element. This chapter describes the architectural and structural factors and, where appropriate, gives a mathematical description. Three basic categories of networks that find use in this work are then described and are then followed by a discussion on the existence of networks suitable for this research.

B. ARTIFICIAL NEURAL NETWORK ARCHITECTURE

The architecture of ANN is described in three levels: topological, data flow, and neurodynamics. The presentation here will follow the guidelines of Neural Computing by NeuralWare, Inc. [NC91], who developed the basic software for this research.

The *topological* level defines the structure of the network and its interface with the environment. The topology also describes how processing elements (PEs) are grouped into layers and how PEs are interconnected to one another. A *layer* here is defined as a group of PEs that act in a similar fashion. In Figure 2.1 a basic schematic of an artificial neural network is presented, consisting of an input layer, an output layer, and one hidden layer. The circles denote the processing elements and the lines denote connections. The term *hidden layer* refers to a layer which does not have a direct connection with the environment. The hidden layers are the essence of the network and extract the features that enable the network to solve nonlinear problems.

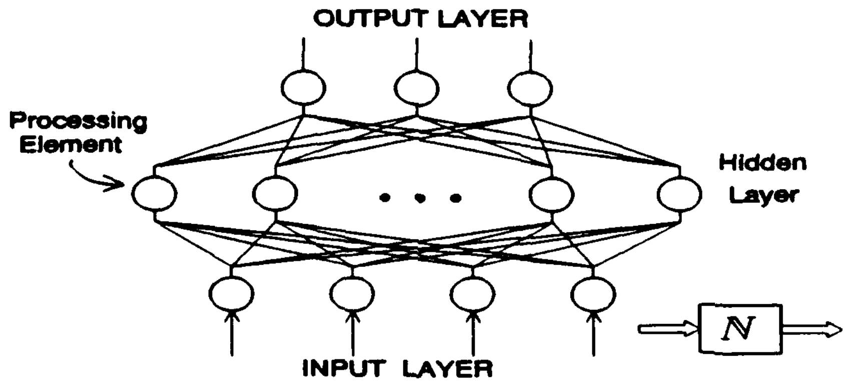


Figure 2.1: **A Basic Schematic of an Artificial Neural Network**

The second level involves *data flow* and describes how information is presented to the network, is passed from one processing element to another, is acted upon within a processing element, and is delivered to the outside world. The data flow level defines the learning rule and the recall or testing process.

The third level, *neurodynamics*, describes in detail the operations that act upon the data within a processing element. This level defines the functions and the behavior of the processing element.

The following paragraphs describe the architectural factors in more detail. The description will start with the environment in which the network operates and will end with a scrutiny of the inner structure of a single processing element.

1. Environment

Recognizing and understanding *the environment* in which the neural network has to perform is as important as choosing the network itself. Most of the time the environment is the factor that dictates the choice of the neural net, its architecture, and its learning scheme. It is important to have a proper interface, including scaling and normalizing the data. In particular, the I/O, its timing and characteristics, determines the number of inputs, outputs, and other criteria such as convergence and pre/post-processing.

After the proper interface on the network is selected, the inner-structure of the network that propagates the data from the input layer to the output has to be chosen. The structure is governed by the propagation scheme.

2. Propagation Scheme

The general structure of the network is controlled by the propagation scheme. The propagation scheme defines how processing elements are connected to one another. Figure 2.1 is an example of connections in a fully connected feed-forward network, from the input elements, through the hidden layer, to the output nodes. The propagation scheme governs the connectivity matrix structurally and by type. The connections may be sequential, by layer, random in order and occurrence, or they may be dictated by the properties of the specific element.

Now that the network is formed, it has to be trained to perform the desired task. The training is done via a learning rule.

3. Learning Rule

Learning is the phase in which the connection weights are adapted according to relation in the input data to some other process. The kind of process categorizes the learning type into one of three categories: supervised learning, unsupervised learning, and reinforcement.

Supervised learning is a process where a desired output corresponds to a given input. Both, input and desired output, should be supplied in the learning phase, during which the weights are updated via some learning rule to match the input-output pair. When the input-output pair is the same vector, the network is referred to as being *auto-associative*. If the desired output and the input vectors are unequal the network is called *hetero-associative*. The work of this thesis uses supervised hetero-associative learning.

Unsupervised learning occurs when no desired output is presented and the network groups the input presentation into some meaningful clusters.

Reinforcement is a process that falls between the two above-mentioned categories. In this case, there is an indicator, a teacher, that indicates whether the response to the presented input is satisfactory.

In any learning category the process is governed by a certain learning rule. This learning rule dictates the change of connections' values or weights to achieve the goal of the training. The parameters that control the learning rule may change while training. The learning phase may need thousands or even millions of presentations until convergence to a satisfactory result is achieved. The learning sequence usually involves a feedback process in which the weights are updated. The specific learning rule that was used in this research, called the back-propagation algorithm, is described in detail in Chapter III.

When the network is trained to satisfactory degree the learning is stopped and the weights in the network are frozen. The network can then be used to perform the task it was trained to. This implementation is governed by the recall phase which is described next.

4. Recall Phase

The *recall* or *testing* phase is represented as a feed-forward pass in the network that gives a response to a certain presentation at the input layer. The recall pass might be a subprocess of the learning phase in which the network output is calculated before the adaptation action is taken due to output error with respect to the desired network response. In essence, after learning is completed, testing is the practical and useful phase.

Recall phase is strongly related to the propagation scheme. To better understand the propagation through the structure, a basic multi-layer feed-forward network is discussed and analyzed next.

C. ARTIFICIAL NEURAL NETWORK STRUCTURE

The behavior of an artificial neural network (ANN) is determined by the network's global and internal structure. This section gives the mathematical description of behavior of an ANN.

The nomenclature adopted here and throughout the dissertation is as follows: Scalars are denoted in upper-case letters. Vectors are denoted in bolded lower-case letters. Matrices are denoted in bolded upper-case letters. Elements of vectors or matrices are denoted with corresponding unbolded lower-case letters with appropriate indexing. Additional indexing, when required, is given and explained in the text.

1. Multi-Layer Neural Network Structure

Multi-layer networks belong to a family of networks consisting of an input layer, output layer and one or more hidden layers. Figure 2.2 shows a basic three layer feed-forward artificial neural network and its block diagram representation. The presented network 'is used for the analysis of the structure and operation. A basic network was selected to simplify description and understanding without the loss of generality in the more complex multi-layer networks. The network shown is fully connected between consecutive layers and a bias term is connected to every processing element.

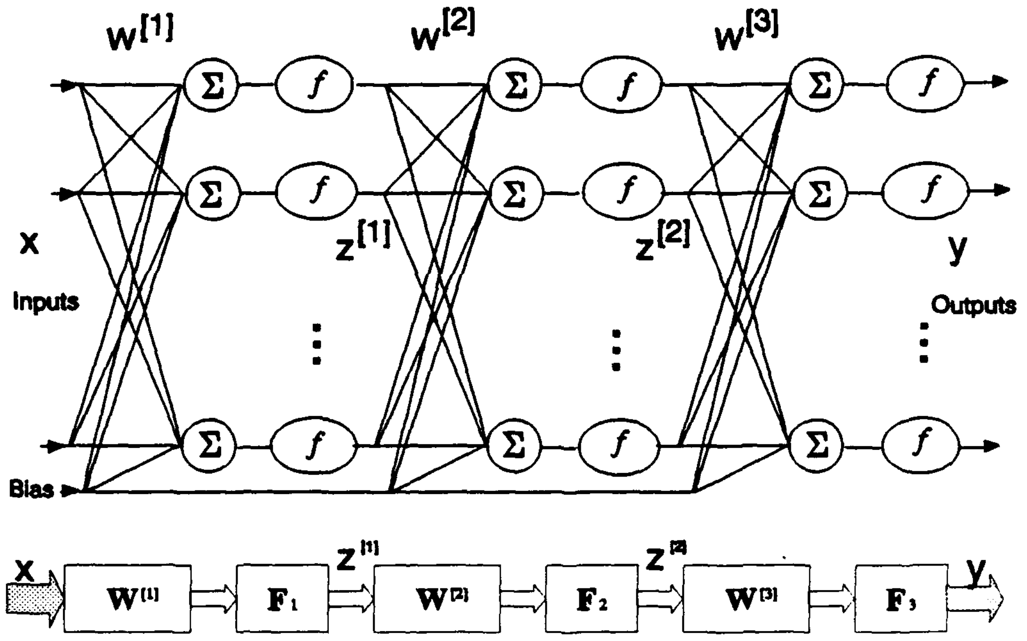


Figure 2.2: **A Three-Layer Feed-Forward Artificial Neural Network**

The network is built from layers that take the same general form. Mathematically a layer is described as

$$y = N(x) \tag{2.1}$$

where N, is a function that maps $x \in \mathbf{R}^n$ to $y \in \mathbf{R}^m$. For a specific layer, r, the nonlinear function $N_r \in \mathbf{R}^n \to \mathbf{R}^m$ is written as

$$N_r(x) = F_r[W^{[r]}x + b^{[r]}] \tag{2.2}$$

where F_r is a diagonal nonlinear operator which represents a set of m nonlinear activation functions $f \in \mathbf{R} \to \mathbf{R}$ that are identical for the current $[r]$th layer [NP91]. The operand of F is the weight matrix $W^{[r]} \in \mathbf{R}^{n \times m}$ times the input vector $x \in \mathbf{R}^n$ summed over all connecting nodes and with a bias term $b \in \mathbf{R}^{m}$. A multi-layer network containing R layers, excluding the input layer, is denoted as $N^{R}_{n_0,n_1,\ldots,n_R}$. Each $[r]^{th}$ layer consists of n_r nodes (processing elements). The network, N, transforms an input vector $x \in \mathbf{R}^{n_0}$ to an output vector $y \in \mathbf{R}^{n_R}$ propagating via (R-1) hidden layers, each consisting of $n_1, n_2, \ldots, n_{R-1}$ nodes, respectively.

For the three-layer feed-forward network shown in Figure 2.2, the output of the network, y, as a function of a given input vector, x, is given by

$$y = N^{3}_{n_0,n_1,n_2,n_3}(x) = F_3\{W^{[3]}F_2[W^{[2]}F_1(W^{[1]}x + b^{[1]}) + b^{[2]}] + b^{[3]}\} = N_1 N_2 N_3(x) \qquad (2.3)$$

The input vector, x, is presented to the network and processed in parallel in each layer. The information is passed in cascade from layer to layer and presented at the output layer as the network's outcome.

Each layer in the network is built from processing elements that work in a similar fashion. The processing elements carry the processing task of each layer. Another step down to the basic element of the ANN is taken next.

2. Processing Element

The *processing element* (PE) is the basic element in any artificial neural network. It has virtually as many inputs as desired and one output. The inputs are summed to create net input for the given PE. A bias is added and the result is acted upon via a scalar *activation function*. The output of the activation function is the output of the PE. A diagrammatic representation of a processing element is given in Figure 2.3. The basic PE configuration just described is used in the present research.

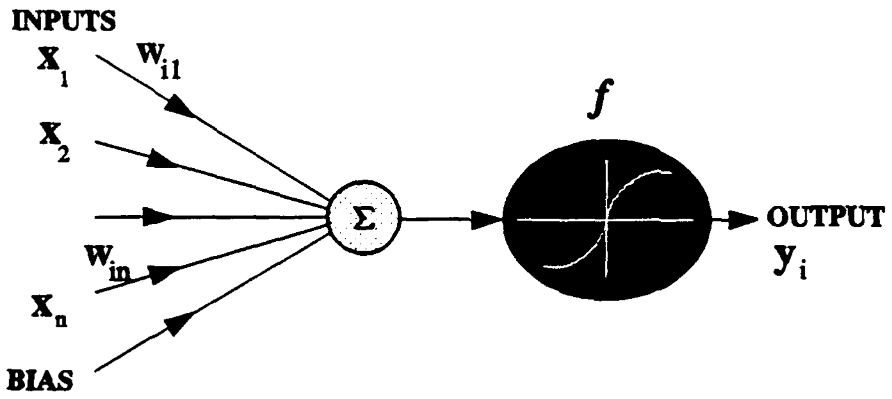


Figure 2.3: **A Basic Configuration of a Processing Element**

The mathematical description of a processing element in the form presented in Figure 2.3 is

$$y_i^{[r]} = f^{[r]}(I_i^{[r]}) \tag{2.4}$$

$$I_i^{[r]} = \sum_{all\ j} x_j w_{ij}^{[r]} + b_i^{[r]} \tag{2.5}$$

where $y_i^{[r]}$ is the output of the i^{th} PE in the $[r]^{th}$ layer and $f^{[r]}$ is the activation function that acts upon the summation value $I_i^{[r]}$. The summation is done over all the outputs x_j of PEs that are connected to the i^{th} PE in the $[r]^{th}$ layer via a weight $w_{ij}^{[r]}$ and the bias $b_i^{[r]}$. It is noteworthy that throughout the work the first subscript of a weight component relates to the target PE whereas the second subscript relates to the source PE.

Variants of a PE's structure may include different summation functions, such as cumulative sum, maximum, minimum, product, etc. The output can either be passed directly or a competitive process between PEs of a given layer can be established

(Kohonen layer). The output value can feed any number of other PEs' inputs, including its own.

An important part in determining the output of a PE is the activation function. Selecting a proper activation function is crucial to successful network training. A more elaborate discussion is given next.

3. Activation Function of a Processing Element

The activation function is a scalar function that acts upon the net sum of the effective inputs. In general, activation functions can be linear, semi-linear or nonlinear functions, and may be non-differentiable. A nonlinear network is achieved by introducing PEs with nonlinear activation functions. In linear processes a linear activation function can yield better results, but generally the more interesting cases are nonlinear. In some cases the weights can even be pre-predicted as will be shown in section C of Chapter IV.

The two activation functions used in this work are the linear function, and the hyperbolic tangent.

The *linear* function, given by

$$f(I) = I \cdot Gain + b \tag{2.6}$$

where I is the summation value multiplied by *Gain* parameter and b is a bias term.

The *hyperbolic tangent*, a continuous monotonic squashing function, which maps the input into a range from minus one to one is given by

$$\begin{gathered} \tilde{I} = I \cdot Gain \\ f(\tilde{I}) = \tanh(\tilde{I}) = \frac{e^{\tilde{I}} - e^{-\tilde{I}}}{e^{\tilde{I}} + e^{-\tilde{I}}} \end{gathered} \tag{2.7}$$

The hyperbolic tangent is differentiable, and moreover, the derivative can be expressed in term of the function itself as

$$f'(I) = (1 + f(I)) * (1 - f(I)) \tag{2.8}$$

The differentiability condition is needed in the application of the back-propagation learning algorithm and an analytical differential as in equation (2.8) saves computations.

Another function commonly used in networks trained via the back-propagation algorithm, and which is similar to hyperbolic tangent, with the difference that it maps to a range from zero to one, is the sigmoid function. The sigmoidal function also shares the property that its derivative can be expressed in terms of the function itself. Figure 2.4, below, shows graphically the functions and their derivatives. Some other activation functions that are used in various neural networks include, but not limited to, sine, signum, step-function and piecewise linear.

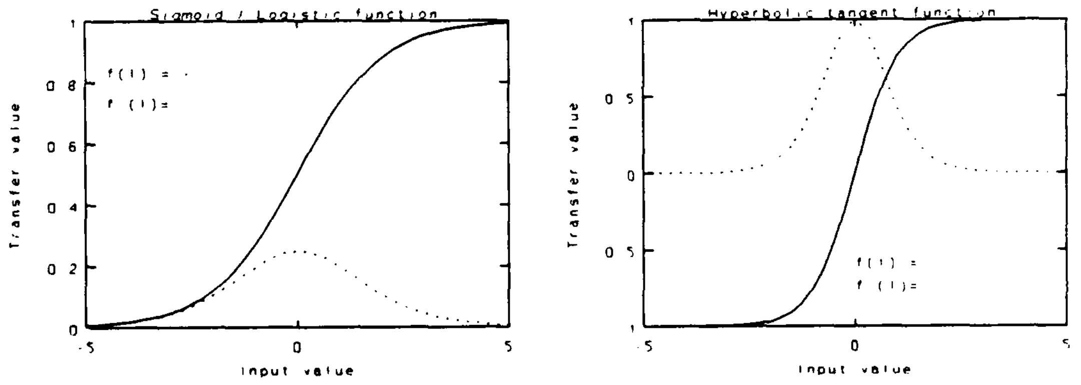


Figure 2.4: **Hyperbolic Tangent and Sigmoid Activation Functions and Derivatives**

Having finished the general structure of an ANN, the specific configuration of the multi-layer networks used in this research are described next.

D. CATEGORIES OF MULTI-LAYER NETWORKS

Multi-layer networks are divided into two main categories. The first and most common is the feed-forward network, as shown earlier in Figure 2.2. The second is the recurrent network. Introducing the recurrent network is essential when dealing with dynamic systems. A block diagram of a recurrent network is shown in Figure 2.5. A multi-layer network may include either or both types. A block diagram of a basic unit of a generalized multi-layer network is shown in Figure 2.6. The principle of combined recurrent and feed-forward network can be further extended to include information from prior input and output vectors, as shown in Figure 2.7. The idea of using general networks in conjunction with dynamic systems, and the detailed structure of the block denoted by $\mathcal{N}$, will become clear in Chapter IV.

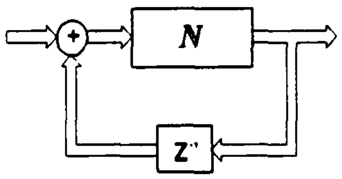


Figure 2.5: **A Recurrent Network**

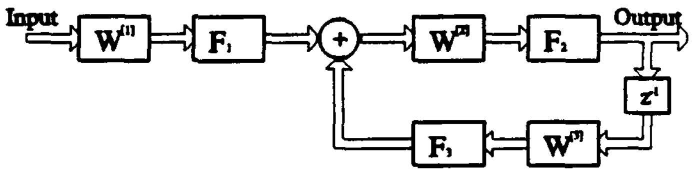


Figure 2.6: **General Multi-Layer Building Block**

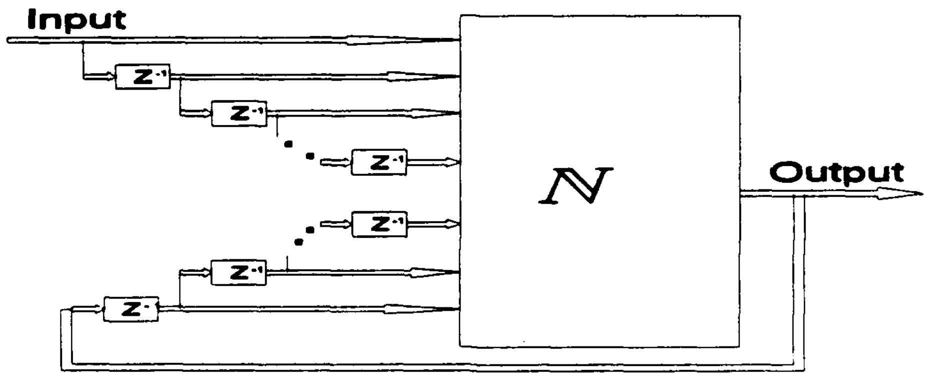


Figure 2.7: **General Regression Network**

E. ARTIFICIAL NEURAL NETWORKS AS GENERAL APPROXIMATORS

Before choosing a network with a specific structure one may ask if there is a network that can do the desired task. The answer to such a question is in the affirmative and is based on two major theorems. The theorems express the existence of artificial neural networks as function mapping devices. ANN is found to be capable of approximating functions from a finite dimensional space to another, to any desired degree of accuracy with only one hidden layer.

The first proof of the capability of a feed-forward multi-layer network to represent continuous functions of many variables by superposition of continuous functions of one variable and addition was given by Kolmogorov's superposition theorem. Hecht-Nielsen entitles the theorem as "Kolmogorov's Mapping Neural Network Existence Theorem" [HN91, pp. 122–124]. The theorem shows the existence of a neural network with linear input and output layers, with as little as one hidden layer, as being capable of mapping any continuous multi-variable function. A corollary gives an upper limit to the number of hidden nodes needed for the mapping ($2n_0+1$, where n_0 is the number of input nodes). However, for every mapping task a *different unknown* activation function is needed. Other expansions of this theorem show the existence of neural networks as mapping devices on different but very wide and practical function spaces [HN91, pp. 131-133].

An important theorem based on Stone-Weierstrass' theorem, given by Hornik, Stinchcombe and White, shows that ANN are general approximators of a bounded function using an *arbitrary squashing* activation function, i.e., a sigmoid or a hyperbolic tangent in an *unknown but large enough* number of PEs to get the desired accuracy [HSW89]. The results show that multi-layer neural networks are universal approximators in mapping any measurable function to a desired accuracy and any lack of success in application must arise from inadequate learning, an insufficient number of hidden PEs, an inappropriate activation function, or the lack of a deterministic meaningful relationship between the input and the desired output.

However, neither theorem really helps to build a "correct" structure of the network, although both show that three layers are enough. Since three-layer network approximators may require an impractical large number of hidden processing elements, in real world problems multi-layer (>3) networks are used to find a tractable solution. The theorems, although not helping in the selection of a proper network structure, do give confidence that an appropriate network exists. Thus, although a suitable network exists, there is no guarantee on the existence of a learning rule to train the weights.

An understanding of the physical nature of the problem may assist in choosing a "good" number of hidden layers and hidden nodes. Until the present, choosing network parameters was a kind of art, usually done by trial and error. A more rigorous way of selecting the number of hidden layers and how many nodes each layer should include is presented in this dissertation as minimal realization of a network (Chapter VI.D). Although the method cannot predict in advance a good choice of number of layers and PEs, the rationale for going through the procedure of finding minimal realization of the network is to achieve the same accuracy with less network complexity. Reassured that a suitable network exists, a suitable training algorithm needs to be chosen. The algorithm used in this work, which has been successfully applied to numerous multi-layer feed-forward networks, is the back-propagation algorithm, described in the next chapter.

III. ANALYSIS OF BACK-PROPAGATION ALGORITHM

Among the many types of learning algorithms, the back-propagation algorithm has been chosen for this research. This learning rule is also called the *generalized delta rule*. The name *back-propagation* is based on the way the errors are handled in changing the weights during learning and is attributed to Rumelhart [RM86, vol. 1, pp. 319–364]. This algorithm is implemented on a hetero-associative network. The standard back-propagation learning rule is derived, followed by four common variations. An extension of the standard algorithm is also presented, with some new highlights used in this research.

A. ARCHITECTURE

Multi-layer networks based on the back-propagation algorithm are feed-forward networks. They consist of an input layer, an output layer, and at least one hidden layer. A typical fully connected feed-forward network with two hidden layers is presented in Figure 2.2. To keep the discussion general, the rest of the derivation will not refer specifically to a particular number of hidden layers or hidden nodes. The representative network shown in Figure 2.2 is merely for demonstration and ease of explanation and does not affect the generality of the back-propagation algorithm.

The back-propagation learning algorithm is used to train multi-layer networks to map or approximate a process or function $y^d = \mathcal{F}(x)$; $\mathcal{F} \in \mathbf{R}^{m \times n}$, represented by input-output pairs $[x, y^d]_p$, where d denotes desired values and p stands for different presentations. An analytical representation of the function may not be available. After the training is complete to a desired accuracy the weights are fixed and the network then generates an approximation of the desired function, y, for a given input, x, as $y = \mathcal{N}(x)$. The rationale for this method of adaptation is given in the next section and the basic form of the learning rule is derived. Many variations have been suggested, mainly to increase convergence speed and to avoid reaching local minima.

B. MINIMIZING GLOBAL MEASURE

The goal of the learning process is to minimize a global measure by modifying the weights in the network. This measure is usually taken as a function of the error at the output of the system. The global measure can in some cases be interpreted in terms of the energy content of the system. Minimizing a global quadratic error is well treated in the literature and is based on the gradient descent method, sometimes referred as the MIT rule, e.g., [BH75, pp. 237-240] or [LL87, pp. 282-288]. The discussion here will concentrate on the discrete domain where a continuous domain equivalence also exists [NP91].

Let a measure, J, as a function of the parameter space, $\boldsymbol{\theta}$, be defined as

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{k=1}^{N} L[\boldsymbol{e}(\boldsymbol{\theta},k)] \tag{3.1}$$

where k is the discrete time indicator and N is the number of samples taken. Among many sensible error functions, L, that can be used, a commonly used one that acts on a network's output error, $\boldsymbol{e}$, is the global quadratic error, defined as

$$L[\boldsymbol{e}(\boldsymbol{\theta},k)] = \frac{1}{2}\left\|\boldsymbol{e}(\boldsymbol{\theta},k)\right\|^2 = \frac{1}{2}\boldsymbol{e}^T\boldsymbol{e} \tag{3.2}$$

Minimization of the measure J with respect to the parameters θ, is done by going in the negative direction of the gradient of J in the parameter space. Hence, the change in the parameters is proportional to $-\nabla_\theta J$ such that,

$$\theta_{new} = \theta_{old} + \Delta\theta \quad , \qquad \textit{and} \tag{3.3}$$

$$\Delta\theta = -\eta \cdot \nabla_\theta J \tag{3.4}$$

where the gradient is defined as $\nabla_\theta J \triangleq \frac{\partial J}{\partial \theta}$, and η is a constant of proportionality. Utilizing the gradient descent method to minimize J, the gradient is determined from equation (3.1) using the chain rule

$$\nabla_\theta J = \frac{1}{N}\sum_{k=1}^{N} \frac{\partial L[e(\theta,k)]}{\partial e(\theta,k)} \cdot \frac{\partial e(\theta,k)}{\partial \theta} \tag{3.5}$$

For optimizing dynamical systems best results are achieved when $N \to \infty$, but in practice, N is chosen over a finite interval, T, $[k-T+1,k]$. In static processes $N=1$ is sufficient [HN91, pp. 136] and usually is the case implied in the back-propagation algorithm. Using the global error function as in equation (3.2), letting i represent the instantaneous time, the performance measure takes the form

$$J(\theta,k) = \frac{1}{T}\sum_{i=k-T+1}^{k} e(\theta,i)^T e(\theta,i) \ , \tag{3.6}$$

and the gradient is

$$\nabla_\theta J = \frac{1}{T}\sum_{i=k-T+1}^{k} e(\theta,i)^T \left[\frac{\partial e(\theta,i)}{\partial \theta}\right] \tag{3.7}$$

The parameters, θ, are then updated every T samples. A single parameter, θ_j, at time $t=(k+1)$, is updated as

$$\theta_j(k+1) = \theta_j(k-T+1) - \eta\frac{1}{T}\sum_{i=k-T+1}^{k} \left[\frac{\partial e(\theta,i)}{\partial \theta_j}\right]^T e(\theta,i) \tag{3.8}$$

The above equation implies that the gradient method can be implemented if the value $\frac{\partial e(\theta,i)}{\partial\theta}$ can be computed on-line for all values of i in the interval $[k\text{-}T+1,k]$.

The major disadvantage of the gradient descent method is the slow convergence rate near the optimum [BH75, p. 238].

The next section shows how this kind of parameter optimization is applied to the back-propagation algorithm.

C. BACK-PROPAGATION LEARNING RULE DERIVATION

Suppose the network in consideration is from the class $\mathcal{N}^R_{n0,n1,\dots,nR}$, as defined in section II.B.5, where the input vector is $\boldsymbol{x}=[x_1,x_2,\dots,x_{n0}]$, and the output vector is $\boldsymbol{y}=[y_1,y_2,\dots,y_{nR}]$. The parameters in the network are the weights $\boldsymbol{W}$, which consist of all relevant $w_{ij}^{[r]}$; $r=1,2,\dots,\mathrm{R}$, as defined in section II.B.6. The training sets are $[\boldsymbol{x},\boldsymbol{y}^d]_p$ for every presentation, p, and $\boldsymbol{y}^d$ has the same dimension as $\boldsymbol{y}$ in order to train all the outputs of the network. The network propagates the input vector to produce an output vector, $\boldsymbol{y}$, which is compared with the desired vector, $\boldsymbol{y}^d$ and result in an error vector, $\boldsymbol{e}$, where $e_m=y_m{}^d\text{-}\ y_m$; $m=1,2,\dots,n_{\mathrm{R}}$. If there is no difference, no learning takes place. Otherwise, the weights are changed in the direction of the negative gradient, $-\nabla_{w_{ij}}E$, to reduce this difference, i.e., the method of steepest descent. The performance measure is defined here as E, in correspondence with the nomenclature used in the neural-networks related literature, corresponds to the total-error energy in the system, in the mean square sense. In accordance to equation (3.6) the measure is given as

$$E = \frac{1}{N}\sum_{p=1}^{N} E_p \quad , \qquad and \tag{3.9}$$

$$E_p = \frac{1}{2} e^T e = \frac{1}{2}\sum_{m=1}^{n_R} (y_m^d - y_m)^2 \tag{3.10}$$

where E_p is the error energy in a single presentation p, over all the n_R output PEs. The dependance of the measure, E, on the weights, W, is through the output y. The change in the weights as in equation (3.4), is proportional to the negative gradient,

$$\Delta w_{ij}^{[r]} = \eta \cdot \left[- \frac{\partial E}{\partial w_{ij}^{[r]}} \right] = -\eta \frac{1}{N}\sum_{p=1}^{N} \frac{\partial E_p(W)}{\partial w_{ij}^{[r]}} \tag{3.11}$$

The coefficient of proportionality, η, is called the *learning rate*. In the batch update, the number of presentation, N, is to be chosen according to the problem, and does not necessarily have to include all presentations. Furthermore, in practice a small finite number is taken, usually N=1, i.e., an immediate update.

For simplicity, without affecting the generality of the mathematical calculations, cascading fully connected layers will be assumed. For clarity, the governing equations for data propagation, equations (2.3) and (2.4), are repeated here.

$$I_i^{[r]} = \sum_j w_{ij}^{[r]} z_j^{[r-1]} \tag{3.12}$$

$$z_i^{[r]} = f^{[r]}(I_i^{[r]}) \tag{3.13}$$

The value z, represents output of hidden PEs. At the input layer $I_i^{[0]} \equiv x_i$, and at the output layer $z_i^{[R]} \equiv y_i$. The error at the output layer is given by

$$e_m = y_m^d - y_m \tag{3.14}$$

The gradient in equation (3.11) is derived assuming that N=1. The end result will be generalized for any N.

Since any functional dependence of E_p on $w_{ij}^{[r]}$ must be through $I_i^{[r]}$, one can write the gradient using the chain rule as

$$\frac{\partial E_p}{\partial w_{ij}^{[r]}} = \frac{\partial E_p}{\partial I_i^{[r]}} \cdot \frac{\partial I_i^{[r]}}{\partial w_{ij}^{[r]}} \qquad (3.15)$$

Define a local error as,

$$\delta_i^{[r]} = -\frac{\partial E_p}{\partial I_i^{[r]}} \qquad (3.16)$$

Substituting equation (3.16) and using equations (3.12) and (3.14) in equation (3.15) gives the negative gradient as

$$-\frac{\partial E_p}{\partial w_{ij}^{[r]}} = \delta_i^{[r]} \cdot \frac{\partial}{\partial w_{ij}^{[r]}} \left[\sum_{q=1}^{n_{r-1}} w_{iq}^{[r]} z_q^{[r-1]} \right] = \delta_i^{[r]} \cdot z_j^{[r-1]} \qquad (3.17)$$

where $z_j^{[r-1]}$ is the output signal of the jth PE in the [r-1]th layer in the forward pass of the pth training presentation.

Evaluating the local error term δ, and using equation (3.13) in the derivation gives,

$$\delta_i^{[r]} = -\frac{\partial E_p}{\partial I_i^{[r]}} = -\frac{\partial E_p}{\partial z_i^{[r]}} \cdot \frac{\partial z_i^{[r]}}{\partial I_i^{[r]}} = -\frac{\partial E_p}{\partial z_i^{[r]}} f^{[r]\prime}(I_i^{[r]}) \qquad (3.18)$$

where f' is the derivative of the activation function with respect to its argument, evaluated at $I_i^{[r]}$. This illustrates the need for the existence of the derivative term $\frac{\partial E_p}{\partial z_i^{[r]}}$.

At the output layer $z_i^{[R]} = y_i$, then

$$\frac{\partial E_p}{\partial z_i^{[R]}} = \frac{\partial E_p}{\partial y_i} = -(y_i^d - y_i) = -e_i\ , \qquad \textit{and} \qquad (3.19)$$

$$\delta_i^{[R]} = e_i f^{[R]\prime}(I_i^{[R]}) \qquad (3.20)$$

At layers other than the output layer, i.e., hidden layers, use of the multidimensional chain rule gives

$$\frac{\partial E_p}{\partial z_i^{[r]}} = \sum_{q=1}^{n_{r+1}} \frac{\partial E_p}{\partial I_q^{[r+1]}} \cdot \frac{\partial I_q^{[r+1]}}{\partial z_i^{[r]}} \tag{3.21}$$

Using equations (3.12) and (3.18) in equation (3.21) yields

$$\frac{\partial E_p}{\partial z_i^{[r]}} = -\sum_{q=1}^{n_{r+1}} \delta_q^{[r+1]} \cdot w_{qi}^{[r+1]} \quad , \qquad \textit{and} \tag{3.22}$$

$$\delta_i^{[r]} = f^{[r]\prime}(I_i^{[r]}) \cdot \sum_{q=1}^{n_{r+1}} \delta_q^{[r+1]} \cdot w_{qi}^{[r+1]} \tag{3.23}$$

In essence, no particular restriction was placed on the error function, E. The only requirement is that the activation function will be differentiable at least once. A recursive mechanism to back-propagate the error at the output layer, equation (3.20), to the hidden layers, equation (3.23), is formed. Altogether, it gives a straightforward learning rule, where according to equations (3.3), (3.8) and (3.11),

$$w_{ij_{new}}^{[r]} = w_{ij_{old}}^{[r]} + \eta \delta_i^{[r]} z_j^{[r+1]} \quad ; \qquad w_{ij}^{[r]}(0) = w_{ij_0}^{[r]} \neq 0 \tag{3.24}$$

where,

$$\delta_i^{[r]} = f^{[r]\prime}(I_i^{[r]}) \cdot \sum_{q=1}^{n_{r+1}} \delta_q^{[r+1]} \cdot w_{qi}^{[r+1]} \qquad \forall\ 0 < r < R, \tag{3.25}$$

and for the output layer,

$$\delta_i^{[R]} = f^{[R]\prime}(I_i^{[R]}) \cdot (y_i^d - y_i) \tag{3.26}$$

In the general case where $N \geq 1$, equation (3.24) becomes

$$w_{ij_{new}}^{[r]} = w_{ij_{old}}^{[r]} + \eta \frac{1}{N} \sum_{p=1}^{N} \delta_{pi}^{[r]} z_{pj}^{[r-1]} \tag{3.27}$$

and equations (3.25), (3.26) are with addition of subscript p on all variables.

Figure 3.1 [NP91, p. 254] gives a diagrammatic representation of the process including the forward and backward passes.

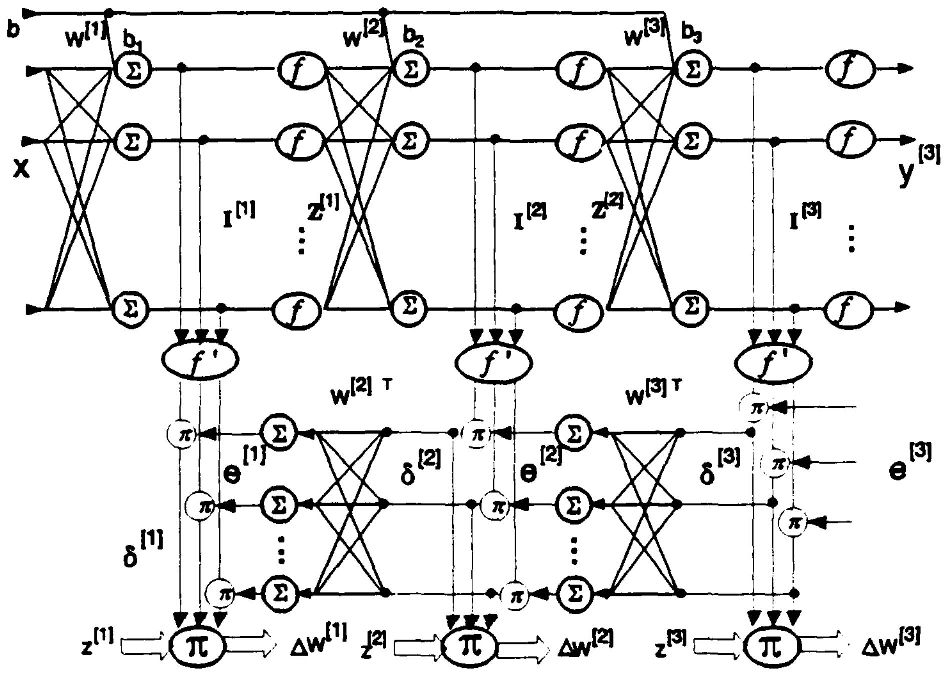


Figure 3.1: A Diagrammatic Representation of Back-Propagation Architecture

As shown, in the backward pass the first derivative of the activation function is used as well as the value of the function itself. For the sigmoid and hyperbolic tangent activation functions, in addition to their desired behavior emphasized in section II.B.7, their derivatives can be expressed in terms of the function itself. Representing the derivative by the value of the function itself saves in the number of calculations needed, and it may also enable a much simpler hardware implementation of a multi-layer, feed-forward neural network trained via back-propagation algorithm. For example, using hyperbolic tangent activation function, equations (2.7) and (2.8), in equations (3.25), (3.26) gives

$$\delta_i^{[r]} = (1+z_i^{[r]})(1-z_i^{[r]}) \cdot \sum_{q=1}^{n_{r+1}} \delta_q^{[r+1]} \cdot w_{qi}^{[r+1]} \quad , \qquad \textit{and} \tag{3.28}$$

$$\delta_i^{[R]} = (1+y_i)(1-y_i) \cdot (y_i^d - y_i) \tag{3.29}$$

respectively.

D. COMMON VARIATIONS ON THE STANDARD BACK-PROPAGATION ALGORITHM

A difficulty with gradient descent-based algorithms, including back propagation, is choosing a suitable learning rate - η. Updating the weights by means of equation (3.24) is based on the assumption that the error surface is locally linear with respect to the step size. At points in which the error surface is "highly" curved, too big a step size may cause the process to become unstable and diverge. On the other hand, keeping the learning rate small yields a very slow convergence, and the process may get trapped in a local minima. Many methods have been developed to overcome this phenomenon and create a more stable update process and much faster convergence. Some of the simple and effective methods are briefly summarized here. [NC91, pp. 94-99]

1. Momentum Term

A momentum term is introduced in equation (3.27) which gives significance to the previous change in the weight. The equation with the momentum term is,

$$\begin{aligned} w_{ij}^{[r]}(k) &= w_{ij}^{[r]}(k-1) + \Delta w_{ij}^{[r]}(k) \\ \Delta w_{ij}^{[r]}(k) &= \eta \delta_i^{[r]}(k) z_j^{[r+1]}(k) + \alpha \Delta w_{ij}^{[r]}(k-1) \end{aligned} \tag{3.30}$$

where $0 < \alpha < 1$ is a scalar momentum term between zero and one. Introducing the momentum term enables faster learning with low learning rate, η, and also allows an increase in η and α before the learning process goes unstable.

2. Cumulative Update of Weights

Implementing equation (3.27), without the normalizing term ($^1/_N$), (with or without the momentum term) also improves the convergence rate, by the fact that Δ weights in the wrong direction are averaged out.

This approach may lead to inadequate learning in some data sets. This is because for well-behaved learning, the training sets have to be well randomized. If the data comes from more than one source, i.e., multiple-function mapping, desirable changes might cancel each other.

3. Fast Back-Propagation

Another variation on the standard back propagation that greatly improves the convergence rate, although it is more sensitive to divergence, is the addition of the error to the PE value.

$$\Delta w_{ij}^{[r]} = \eta \cdot \delta_i^{[r]} \cdot (z_j^{[r]} + \beta \delta_j^{[r-1]}) \tag{3.31}$$

where $0 \leq \beta < 2$ is a factor, which when set to zero yields the standard back-propagation algorithm. It can be seen that asymptotically when $\delta \to 0$ (as training progresses), this learning rule approaches the basic back-propagation algorithm. The momentum term as discussed in section *1* may also be applied.

4. Modifying the Derivative

As seen in Figure 2.5, at the extremes where the activation function becomes saturated, the derivative of the activation function goes asymptotically to zero, and thus the PE stops learning, since $\Delta w \propto f'$. By adding a small offset to f', the PE continues learning even when it is saturated. This can in turn speed up the learning.

Another significant variant of the back-propagation algorithm, called back-propagation through the plant, that treats the problem where the desired values at the output layer are not known directly, is presented in the next section. It should be mentioned that any of the four variants described above can be further applied to the new algorithm.

E. BACK PROPAGATION THROUGH THE PLANT

The following training algorithm was developed in the context of controlling a plant with only qualitative knowledge on the plant behavior by Saerens and Soquet [SS89]. An extension of the algorithm is proposed here to give a more accurate implementation of the gradient descent, and results in faster convergence. The change comes in estimating the Jacobian of the plant, and keeping the process within the domain, an important issue when dealing with non-linear systems.

1. Architecture and Formulation

As was shown, in order to train a network, the pair of input and corresponding desired output values need to be supplied. There are instances where the neural network is cascaded to another physical process, and the ANN output is connected to the process or plant input, as shown in Figure 3.2. Suppose the network in consideration is from the class $N^R_{n0,n1,\dots,nR}$, as defined in section II.B.5, where the input vector is $x=[x_1,x_2,\dots,x_{n0}]$, and the network output vector is $u=[u_1,u_2,\dots,u_{nR}]$. The parameters in the network are the weights W, which consist of all relevant $w_{ij}^{[r]}$; $r=1,2,\dots,R$, as defined in section II.B.6. The output of the network u, is used to excite a non-linear dynamic process, Φ. The output of the non-linear process is given by $y=[y_1,y_2,\dots,y_S]$, where S is the number of outputs of Φ. The training sets are $[x,y^d]_p$ for every presentation, p. The formulation of the combined system, with special attention to the indexing used in the derivation of the learning rule, is given by

$$\begin{cases} y(k+1) = \Phi\,[\,u(k),\, v(k)] \\ u(k) = N^R_{n_0,n_1,\dots,n_R}[x(k),\, W(k)] \end{cases} \tag{3.32}$$

where $v(k)$ are internal parameters of Φ that do not depend on the current input $u(k)$, but may depend on previous inputs, W represents the weights in the network, and

$$\begin{aligned} u(k) &= \{u_l(k),\ l=1,\dots,n_R\} \\ y(k) &= \{y_m(k),\ m=1,\dots,S\} \end{aligned} \tag{3.33}$$

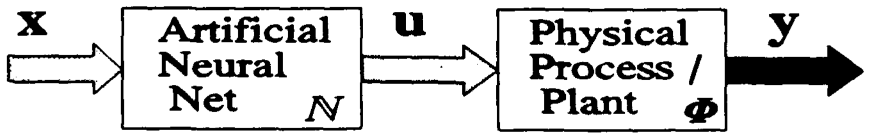


Figure 3.2: Artificial Neural Network preceding a Physical Process

The desired values are known at the output y, but not at the intermediate point u, which are the values needed for training a hetero-associative neural network. In cases like this, a variant of the back-propagation algorithm is used, called back propagation through the plant (BPTP). The idea is to transfer the error information at the output of the plant, through the plant, to the output of the network u. There are two ways of doing this.

In the first a neural network is used to emulate the plant or process, and the weights are frozen after training. A new neural network is built and connected in precedence to the emulated process. The combined network is trained using regular back propagation. The error at the output y is transferred backwards through the fixed weights of the emulated plant. The transferred error is further back propagated through the preceding network. The weights in the preceding network are updated until a small enough error is achieved at the output of the emulated plant.

The second method avoids the need to have a neural network representation of the process/plant. The error at the output of the plant y, is transformed to a corresponding error at the output of the network u. From that point it is possible to use the standard back-propagation algorithm. The procedure of back-propagating the error through the plant is derived in the next section.

2. Derivation of Learning Rule for Back-Propagation Through Plant

As before, the objective of the training is to minimize an error function that acts upon the output values due to changes in the weights. Define a measure on the output as in equation (3.10) as $E_p = \frac{1}{2}\sum_{m=1}^{q}(y_m^d - y_m)^2$. The parameter space is $\boldsymbol{W}$, where $w_{ij}^{[r]}$ is the weight connecting the output of the jth PE in source layer $[r\text{-}1]$ to the input of the ith PE in target layer $[r]$. The development will follow the procedure set in subsection C. The change in the weights is proportional to the negative gradient of the measure with respect to the weights as in equation (3.11) $\Delta w_{ij}^{[r]} \propto -\left(\nabla_{w_{ij}} E_p\right) = -\eta \dfrac{\partial E_p}{\partial w_{ij}^{[r]}}$.

Following the exact derivation as in equations (3.15)-(3.18), gives

$$-\frac{\partial E_p}{\partial w_{ij}^{[r]}} = \delta_i^{[r]} z_i^{[r-1]} \;, \qquad \textit{and} \tag{3.34}$$

$$\delta_i^{[r]} = -\frac{\partial E_p}{\partial z_i^{[r]}} \cdot f^{[r]\prime}(I_i^{[r]}) \tag{3.35}$$

The partial derivative of the measure with respect to the output of each PE needs to be evaluated. Starting at the output layer, using a multidimensional chain rule and utilizing equations (3.10), (3.11) and (3.32), noting that at the output layer $z^{[R]} = \boldsymbol{u}$, gives

$$\frac{\partial E_p}{\partial u_l} = \sum_{m=1}^{s} \frac{\partial E_p}{\partial y_m} \cdot \frac{\partial y_m}{\partial u_l} = -\sum_{m=1}^{s} e_m J_{\Phi_{ml}} \tag{3.36}$$

where J_Φ is the Jacobian of the function Φ with dimensions $s \times n_R$, and the (m,l) element is given by $\frac{\partial y_m}{\partial u_l}$. Substituting equation (3.36) into (3.35) yields

$$\delta_l^{[R]} = f^{[R]\prime}(I_l^{[R]}) \cdot \sum_{m=1}^{s} e_m J_{\Phi_{ml}} \tag{3.37}$$

and therefore,

$$\Delta w_{lj}^{[R]} = \eta\, \delta_l^{[R]} z_j^{[R-1]} \tag{3.38}$$

which has the same form as equation (3.24) in the generalized data rule, and equation (3.26) is replaced by equation (3.37) as the back-propagated error at the output layer.

Following on with the derivation, equations (3.21)-(3.23) give the same results, i.e., equation (3.25).

Tne BPTP learning rule is a recursive procedure identical to the standard back-propagation algorithm, with the modification on how the error at the output layer is calculated. The same result under different derivation can be found in [SS89]. Recognizing in the classical back-propagation scheme that $\Phi(u) \equiv \mathbf{I}_{n_R} \cdot u$, and $J_\Phi = \mathbf{I}_{n_R}$, where $\mathbf{I}_{n_R}$ is the identity matrix of size n_R, equation (3.37) reduces to (3.26) exactly. This shows that the BPTP variant is a direct extension of the basic rule, or rather, the basic rule is a particular case of the former. Hence all the other variants described in subsection D apply to back propagation through the plant as well.

In the derivation, the only parameter that does not have a closed from for the general non-linear case, is the Jacobian. Researchers treat it in different ways, and another approach to estimating the Jacobian is proposed next.

3. Estimating the Jacobian of the Plant

Close examination of equation (3.37) leaves the Jacobian J_Φ a process dependent parameter, that not always has a closed form formulation. Jordan [JI89] suggested a way to calculate an estimation of the Jacobian using a neural network emulation of $\Phi(u)$. Saerens and Soquet [SS89] showed that if no details are available on the Jacobian, knowing its sign might be sufficient, given this is *a priori* known. Because this is a rough approximation, convergence is rather slow, since the descent is not in the direction of the gradient. In nonlinear systems, monotonic behavior is not guaranteed. In off-line procedures, where the process $\Phi(u)$ is simulated, if the exact value were unknown or were difficult to calculate, a simple numerical differentiation may be used. A similar technique was used by Psaltis, Sidris and Yamamura [PSY88]. As was mentioned earlier, the exact value is not necessary [SS89], but a close value is preferable for more rapid convergence. Thus, an "influence coefficient" may be considered instead of the Jacobian. Assuming Φ is well behaved at the point of calculation for a small enough Δu, backward or forward differences are sufficiently accurate and simplify the calculations in computing an estimate of the Jacobian. The choice of backward or forward differences is based on the previous state of $\boldsymbol{u}$ in order to ensure staying within the domain. The influence coefficients are calculated at every iteration for every state in the vector $\boldsymbol{u}$. Denoting the iteration sequence by the index k, the choice of difference method is as simple as

if $u_l(k) > u_l(k+1)$, use forward differences.

if $u_l(k) < u_l(k+1)$, use backward differences.

The calculation is done for each state in $\boldsymbol{u}$ sequentially, leaving the rest constant.

The influence coefficients are calculated as follows:

Using forward differences

$$\tilde{J}_{m,l}(k) = \frac{\Phi_m(u_1,u_2,\ldots,u_l+\Delta u_l,\ldots,u_{n_s},k)-\Phi_m(u_1,u_2,\ldots,u_l,\ldots,u_{n_s},k)}{\Delta u_l} \tag{3.39}$$

Using backward differences

$$\tilde{J}_{m,l}(k) = \frac{\Phi_m(u_1,u_2,\ldots,u_l,\ldots,u_{n_s},k)-\Phi_m(u_1,u_2,\ldots,u_l-\Delta u_l,\ldots,u_{n_s},k)}{\Delta u_l} \tag{3.40}$$

Where $\Phi_m(\cdot,k)=y_m(k)$, and Δu_l is the perturbation in the u_l state.

The implementation of artificial neural networks trained via the back-propagation algorithm is described next. The implementation makes use of the methods described in this chapter, on feed-forward and recurrent networks structure, described in Chapter II.

IV. NEURAL NETWORK REPRESENTATION OF DYNAMICAL SYSTEMS

It is interesting to examine how neural networks can interact with, act upon, or represent dynamical systems and processes. It seems evident that neural networks can have an important role in dealing with dynamical systems. There are already numerous examples of the application of neural networks to dynamical system problems, and some examples can be seen in references [MSW90], [NP90], [NP91], [SS89], [HWB92], [SM92]. This research deals with the emulation and control of non-linear dynamical systems or processes using artificial neural networks. In this chapter a class of non-linear multi-input multi-output dynamical systems is developed. The affiliation of such systems with neural networks is then presented, followed by the special case of a linear time-invariant system. It is shown that a single network can represent a multitude of systems from the same class. A brief survey of previous work in related topics is presented, as well as motivations for extensions in this work.

Narendra and Parthasarathy have presented extensive work, [NP90] and [NP91], on dynamical systems and their neural-network representation. They dealt with Auto Regressive Moving Average with eXternal input (ARMAX) model structures and mainly with single-input single-output (SISO) systems. Based on the models described in references [NP90] and [MSW90, pp. 126–128], another presentation is suggested for multi-input multi-output (MIMO) state-space models.

It is possible to transform ARMAX form to a state-space form, since state-space representation is preferable. State-space representation is especially useful in giving insights into the physical mechanisms of the system, which can usually be more easily incorporated into state-space models than into ARMAX models [LL87, p. 81]. Another reason that working in state-space became very popular, e.g., in modern control theory, is that state-space models are more reliable in computer analysis [GLLT90, pp. 1-67,1-68], since the numerical algorithms are more stable. In fact, for MIMO systems, the state-space representation is the only convenient model.

Most physical systems are more easily described in the continuous time domain, simply because most laws of physics are thus expressed. However, in many real-life implementations, a discrete representation is used and the modeling developed here will be based on a discrete representation. A similar description for the continuous time is achieved by replacing the difference equation by differential equations.

A. CHARACTERIZATION OF NONLINEAR DYNAMICAL SYSTEMS

Four classes of unknown nonlinear discrete dynamical systems were suggested by Narendra and Parthasarathy [NP90]. The models are extended here for MIMO dynamical systems with the addition of an output measurement equation. Common to all of them is that the state at $t=(k+1)$ is derived directly from the state and excitation functions at time $t=k$. If previous states are needed to describe the system, the prior states are augmented, forming a new state vector. The models are assumed to have the measured output of the system as a linear function of the state vector.

The nomenclature adopted here uses symbols that are usually used in control theory. In addition, upper-case italic bold letters followed by brackets represent vectorial functions, subscripted with l for a linear function, and with nl for a non-linear function. Upper-case bold letters with or without succeeding parentheses, represent matrices. The state vector is denoted as $\boldsymbol{x}$, the measured output vector is denoted as $\boldsymbol{y}$, the input sequence is $\boldsymbol{u}$, the discrete time is represented by k, and $\boldsymbol{\theta}$ are the unknown parameters of the system.

1. Model I

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{F}_l[k,\boldsymbol{\theta}_f]\,\boldsymbol{x}(k) + \boldsymbol{G}_{nl}[k,\boldsymbol{\theta}_g,\boldsymbol{u}(k)] \;; \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \\ \boldsymbol{y}(k) = \boldsymbol{C}(k)\,\boldsymbol{x}(k) + \boldsymbol{D}(k)\,\boldsymbol{u}(k) \end{cases} \tag{4.1}$$

In this nonlinear model, the state vector depends linearly on its prior state and nonlinearly on the input signals. The output is a linear combination of the state and input vectors. This model is very common in many mechanical systems where the inputs are subjected to nonlinearities such as saturation, backlash (dead zone), and switching (bang-bang).

2. Model II

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{F}_{nl}[k,\boldsymbol{\theta}_f,\boldsymbol{x}(k)] + \boldsymbol{G}_l[k,\boldsymbol{\theta}_g]\,\boldsymbol{u}(k) \;; \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \\ \boldsymbol{y}(k) = \boldsymbol{C}(k)\,\boldsymbol{x}(k) + \boldsymbol{D}(k)\,\boldsymbol{u}(k) \end{cases} \tag{4.2}$$

Here the nonlinear function $\boldsymbol{F}_{nl}$ acts on the previous states, while the inputs are linear. This is a very popular model in systems where kinematics are involved (nonlinear trigonometric functions), systems with viscous friction, e.g., airplanes. This model or its linearized version, is very common in implementations of control actions, $\boldsymbol{u}$, on the system $\boldsymbol{F}$.

3. Model III

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{F}_{nl}[k,\boldsymbol{\theta}_f,\boldsymbol{x}(k)] + \boldsymbol{G}_{nl}[k,\boldsymbol{\theta}_g,\boldsymbol{u}(k)] \ ; \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \\ \boldsymbol{y}(k) = \boldsymbol{C}(k)\,\boldsymbol{x}(k) + \boldsymbol{D}(k)\,\boldsymbol{u}(k) \end{cases} \tag{4.3}$$

In model III, both the input values and the previous state vector are subjected to two separate but additive nonlinear functions. Real-world examples for this model can be simply a combination of examples from models I and II.

4. Model IV

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{H}_{nl}[k,\boldsymbol{\theta}_h,\boldsymbol{x}(k),\boldsymbol{u}(k)] \ ; \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \\ \boldsymbol{y}(k) = \boldsymbol{C}(k)\,\boldsymbol{x}(k) + \boldsymbol{D}(k)\,\boldsymbol{u}(k) \end{cases} \tag{4.4}$$

This is the most general form that represents the current state as a nonlinear function of both the previous states and inputs combined. Although it is the most general, it is the most difficult to deal with, and in practice, approaches described in models I to III are mostly used.

For every suggested model there is a corresponding neural network structure. The structure characterizes the physical behavior of the model, and is trainable by the generalized delta rule. The representations and their generalizations are described in the following section.

B. NEURAL NETWORK REPRESENTATION OF NON-LINEAR DYNAMICAL SYSTEMS

To represent linear or non-linear dynamical systems, recurrent networks must be used. The issue of building a proper recurrent network structure to represent, emulate or identify a non-linear dynamical systems has been treated by researchers in different ways. Narendra and Parthasarathy worked with SISO systems [NP91]. Werbos [WP90] and Gherrity [GM89] discuss training aspects for a recurrent networks. Seidi and Lorenz suggested a general structure by which a recurrent neural network could approximate a non-linear dynamic system, from the general form of model IV, where the measurement equation was a non-linear function of the states only [SL91]. Unfortunately, they did not address the problem of finding a weight adjustment algorithm to insure convergence of the system.

A structure containing hidden recurrent layer, was also investigated by the author, in terms of training via back-propagation method. In the general case, when the output does not have a full presentation of the state vector, successful training via the back-propagation method is not guaranteed. The reason for the inability to learn to a reasonably small error is that during training the values of the state vector are unknown.

The research conducted here classifies dynamical systems in two categories. First by the order of the input and output vectors, and the second by the relation of the output vector to the state vector. Each category is then subdivided as follows:

Category A: Input-Output Orders

1) *Single-input, single-output (SISO).*

2) Single-input, multi-output (SIMO).

3) Multi-input, single-output (MISO).

4) Multi-input, multi-output (MIMO).

Category B: Output to State vectors relation

1) Output vector contains full state representation.

2) Output vector contains partial state representation.

For emulation, the dynamical system has to be completely *reconstructible* (also defined as constructible in the literature) in the linear case [KS72, pp.65-79,462-465], or *realizable* in the non-linear case [IS89, pp.129-144]. An examination shows that systems belonging to category B.1, i.e., systems containing full state output are rather trivial cases to emulate or identify using artificial neural networks. An example of linear time-invariant system with full state output is demonstrated and analyzed later in section IV.C. Systems from categories A.1 and A.2, i.e., single input systems, are also rather straight-forward, since there is a theoretical solution for the one step ahead prediction and control [LL87, pp.51-65], that is easily adapted to neural network implementation. Most examples in the literature deal with models from or converted to categories A.1, A.2 and B.1.

The main thrust of this work was to find a network structure to represent MIMO dynamical systems, *trainable* via back-propagation algorithm. The general form in which an ANN is trained to emulate or identify a plant is presented in Figure 4.1, where the inputs and desired values are generated, either by simulation or experimental data, and presented to the network in parallel. The network is trained until the error at the output reaches an acceptable level. The specific inner structure of the network is now described.

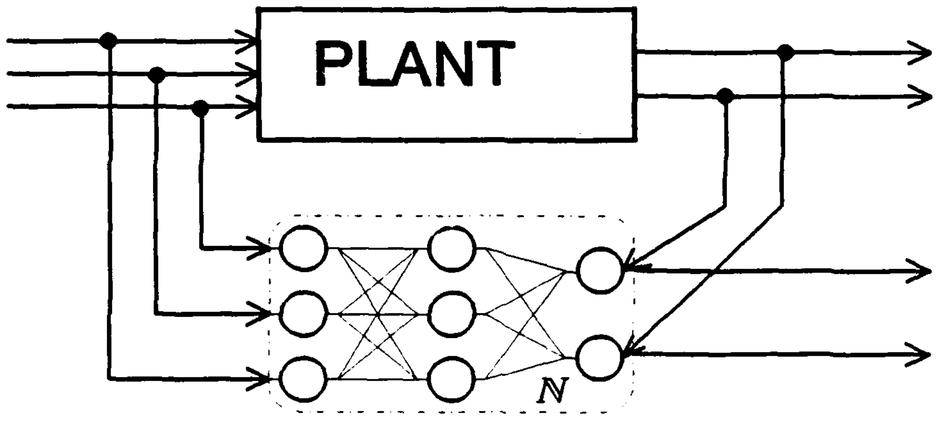


Figure 4.1: A Neural Network Emulation of Dynamical System

In constructing an ANN it is important to take into consideration the structural and trainability aspects, which have to comply with theory. To better explain the structure and operation of an ANN used to emulate a dynamical system, first the solution is introduced followed by the incentives that motivated the approach. A representative network used to emulate dynamical systems represented by models I-III as described in section IV.A is given in Figure 4.2, where the circles denote processing elements, solid lines connecting PEs are adjustable weights, and the dash-double-dot line (—··—) represents a fixed unity-valued weight. The structure is for any MIMO system. A system with three inputs and two outputs was selected for demonstration purposes only. The configuration retains the "black-box" form of the emulated system, by interfacing with the environment only with current values.

1. Structural Aspects

The structural considerations are considered in sequence. Systems from category A.4 - MIMO are considered, noting that categories A.1-A.3 are subsets of the former. The discussion starts with systems from category B.1 - full state output, such that for the measurement matrix C, $rank\,(C) = n_x$, and $n_y \geq n_x$; where n_y and n_x are the dimensions of the output and state vectors respectively. For this case it is possible to talk in terms of y instead of x. For example, model III can be written as

$$y(k+1) = \tilde{F}_{nl}[k, \theta_f, y(k)] + \tilde{G}_{nl}[k, \theta_g, u(k)] \tag{4.5}$$

where $\tilde{F}$, $\tilde{G}$ relates to F and G as y of eq. (4.5) relate to x in eq. (4.3), given that $x(k)$ is the unique solution of $C(k)\ x(k) = y(k) - D(k)\ u(k)$. The unknown parameters of the functions $\tilde{F}$, $\tilde{G}$ are denoted θ_f, and θ_g respectively.

Each of the linear and nonlinear functions presented in models I-IV can be represented by a multi-layer neural network. The functions $\tilde{F}_l$, $\tilde{F}_{nl}$ are mapped by a network from the class $\mathcal{N}^{R_y}{}_{n_y, n_1, \ldots, n_y}$. Similarly, the functions $\tilde{G}_l$, $\tilde{G}_{nl}$ are mapped by $\mathcal{N}^{R_u}{}_{n_u, n_1, \ldots, n_y}$, where n_u is the dimension of the input vector, and $\tilde{H}_{nl}$ is mapped by $\mathcal{N}^{R}{}_{(n_u+n_y), n_1, \ldots, n_y}$. The network combined from feed-forward and the recurrent parts for u and y respectively. In this case,

$$y(k+1) = \mathcal{N}_r[y(k)] + \mathcal{N}_f[u(k)] \tag{4.6}$$

where $\mathcal{N}_r$ is a recurrent network shown on the right of Figure 4.2, while $\mathcal{N}_f$ is a forward network as depicted in the same figure on the left. The network $\mathcal{N}_r$ is an extended form of the Hopfield recurrent network represented in a block diagram in Figure 2.5, where z^{-1} denotes a one-step delay. Note that Hopfield network is from the class $\mathcal{N}^{1}{}_{n,n}$, where the input is fully connected to the output layer and the output values are fed back to the input layer via a one-step delay line.

The more difficult case is for systems from category B.2, where the state variables cannot be computed from the output vector. For systems from category B.2, The measurement matrix C has $rank\ (C) < n_x$, making the determination of N_r more difficult. From linear systems theory it is known that the system needs to be constructible [KT80, pp.90-103], or reconstructible [KS72, p. 67] in order to be able to predict succeeding outputs. In non-linear systems, realizability is required [IA89, pp.131-144]. Reconstructibility means that present output values can be predicted from past values of the output. The need for previous information leads to introducing of *regression layers*.

The inherent structure of the regression layer acts as a delay-line to provide the network with previous measurements of the input and output of the system. The order of a regression layer is defined as the number of previous measurements taken. The order of a regression layer is determined by the system which the network has to emulate, and sometimes may be assigned theoretically (see also minimal realization in Chapter VI.D). Examples of two regression layers of order three are shown in Figure 4.2. On the left side the data from the input layer is regressed, and on the right side of Figure 4.2 the output values are regressed structuring a recurrent network. The regression layers become the practical input layer to the network, even though they are represented as an internal layers. The delayed values can be viewed as additional inputs to the network, to help create a proper relationship between the input and the output of the network.

The network is structured to correspond with the model chosen to represent the system. In the case of models I-III, represented by Figure 4.2, the effect of the input, described by G, is represented by N_f in the left branch of the figure, and the past output influence, described by F is represented by N_r at the right branch. The resemblance to the physical behavior gives more insight to the system when analyzing the trained network and the weight matrices. Another advantage is gained when using the emulated system for training a peripheral network, such as training a controller network for the system. In the given structure, the back-propagated error that appears at the input of the emulated system is due to the input's effectiveness, as opposed to errors spread over a large number of nodes, part of which are not relevant to the system's inputs, as happens when one uses the generalized-delta-rule. From numerous simulations it was found that a network structure similar to the mathematical model yielded better results.

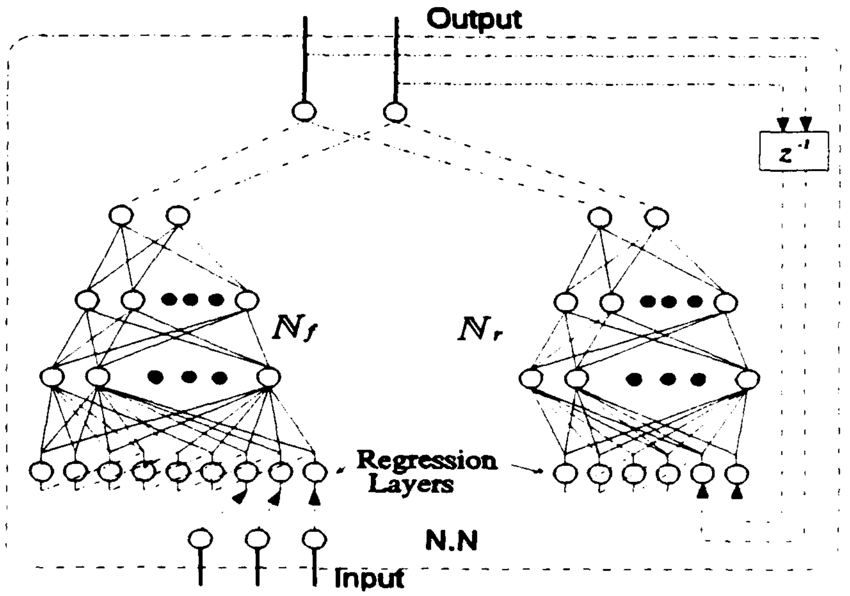


Figure 4.2: Artificial Neural Network Emulator for Models I-III

A suitable network configuration was established based on physical behavior of the system, and theoretical considerations for MIMO dynamical system. A suitable training procedure is next discussed.

2. Trainability Aspects

It was shown in Chapter II.E that an ANN used to approximate a function/process, uses a squashing activation function in the PEs. If a squashing function is used, then the network has bounded output. In a physical system it is usually desirable, and practical also to have bounded inputs. Therefore, a dynamical system that is to be represented by an ANN should normally be a bounded-input bounded-output (BIBO) system over a time interval $[t_0,t_f]$. A system which is BIBO over time interval is denoted as *bounded system* (see definition 3-36 in [AF66, p.101]). Note that a bounded system *does not* necessarily impose BIBO stability. In fact, unstable linear systems do belong to bounded systems, and can be emulated, as shown by Bertrand and Collins in [BC92] and [BD91]. It is worth mentioning that using a squashing activation function is not necessary. Indeed, to represent linear systems, better results are often obtained using a linear activation function [SR89], [BD91].

For bounded systems, two training schemes are possible, *parallel* and *series-parallel* schemes [NP90, pp.13-14]. For example, assume the dynamical system $y^d(k+1) = \boldsymbol{Q}[y^d(k), \boldsymbol{u}(k)]$, where $\boldsymbol{Q}$ is a vectorial bounded function to be emulated by N, with $y(k+1)$ as the output of the network. The two schemes are herein described, using this example.

In the parallel-scheme the current network output is transferred to the output regression layer. The intermediate network approximations are used while training and in the recall phase, resulting in a process of the form

$$y(k+1) = N[y(k), u(k)] \tag{4.8}$$

The parallel-scheme has not been shown to be globally stable, even in the linear case [MSW90, p.130].

The series-parallel scheme, uses the desired / known values, y^d, in the output-regression layer while training, making a simple feedforward network. During the training the procedure takes the form of

$$y(k+1) = N[y^d(k), u(k)] \tag{4.9}$$

Since the input and output values are assumed to be bounded, and there is no feedback loop containing non-linear elements, the standard back-propagation algorithm can be used. After training is complete $y(k) \approx y^d(k)$, the parallel scheme is used in the recall phase, i.e., the network outputs are regressed. The series-parallel scheme has been shown to be stable [MSW90, p.130], and thus it will be used in this research.

This section established a suitable structure and training scheme for an ANN emulation of a non-linear MIMO dynamical system. A simple and unique result arises in the linear time-invariant systems. The power of ANN in identification and emulation of linear systems is demonstrated in the following section.

C. ARTIFICIAL NEURAL NETWORK REPRESENTATION OF LINEAR TIME-INVARIANT SYSTEMS

The linear time-invariant (LTI) system is a special case of either model I, II or III, where both F and G are linear functions. Consider the dynamical system

$$\begin{cases} x(k+1) = Fx(k) + Gu(k)\ ; \quad x(0) = x_0 \\ y(k) = Cx(k) \end{cases} \tag{4.10}$$

where C is full rank and invertible when the number of outputs is equal the number of state variables. That specific system can be represented by a fully connected neural network from the class $N^1_{n_i,n_o}$ with linear activation function for the PEs, where the output layer is fully connected to itself, as seen in Figure 4.3 for an example of a system with four inputs and three outputs.

For any network it is unknown in advance to which values the weights will converge to, and whether the solution is unique. The final results strongly depend on the structure of the network and the initial weight randomization. In the LTI case however, not only is convergence guaranteed but it is also unique; perhaps surprisingly, the weight matrices have a simple physical meaning. Examining the weights after the network converged, gives insight to the behavior of the system. In fact, if the system is known the weights are known a priori, and no training is needed. Using the fact that C is invertible, eq. (4.10) can be written in terms of y only as

$$y(k+1) = CFC^{-1}y(k) + CGu(k)\ ; \quad y(0) = Cx_0 \tag{4.11}$$

In several tests with LTI systems, the weights converged as expected to $W^{[1]}=CG$, and $W^{[2]}=CFC^{-1}$, with residual error of zero if numerical rounding errors of a computer simulation were ignored.

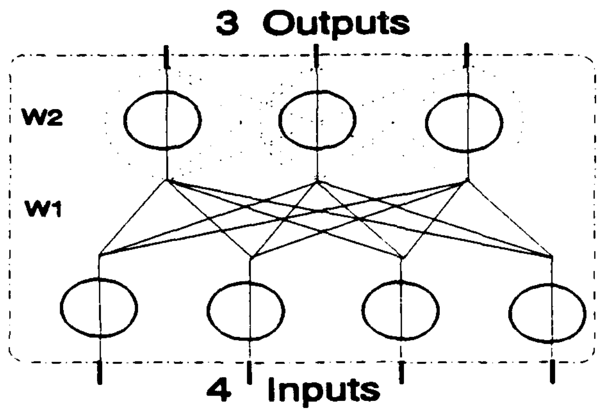


Figure 4.3: A Neural Network Representation of LTI System

D. EMULATION OF MULTIPLE MIMO SYSTEMS IN A SINGLE ANN

Throughout the chapter it was shown that artificial neural networks have the capability of approximating non-linear dynamical systems. A new method is now proposed to extended the capability to emulate multiple systems from the same class in a single network.

The different systems are assumed to have the properties discussed in subsection B. The dynamical systems belong to the same class and have the same input and output order. Additional *type* nodes, as many as the different models, are introduced at the input layer. At the presentation of information of a specific model, the corresponding node takes the value of one while all the rest of the type nodes are valued minus one.

The training is in series-parallel scheme. To ensure proper training of all systems involved, the following conditions were obtained:

1) Random presentation of each system.

2) Balanced presentation of all systems.

3) Persistent excitation in the domain of interest.

The domain of interest in dynamical systems refers to both frequency and amplitude, over which the systems are to be mapped or emulated. The subject of persistent excitation will be discussed in more detail in Chapter VI.B, and is important for the single system as well.

The implementation of the three conditions in this research for different dynamical models was as follows:

- Model is changed every presentation.
- Model number is selected for a random sequence without repetitions.
- Input signal is random-binary-sequence.

The input signal is band limited to the desired spectrum, and amplitude of ± 1, with corresponding scaling of the models.

Until now the discussion concentrated on emulation of dynamical systems by artificial neural networks. The next chapter discuss the role of ANN in adaptive control of dynamical systems.

V. ARTIFICIAL NEURAL NETWORKS IN ADAPTIVE CONTROL

The control of LTI systems is a mature field with well established foundations on conditions and techniques, which define how LTI system can be controlled to yield desired responses. On the other hand, requirements for stable controllers of non-linear and time-varying systems are not generally available. In the last forty years, there has been significant progress in developing methods for identification and adaptive control of linear systems with unknown or time-varying parameters. Adaptive control techniques were based on the assumption of reasonably slow-varying changes in the parameters of the system, and depend on methods from linear estimation and control theory [AW89]. A recent trend of detection abrupt changes [AD91], [DS89], may be introduced to the adaptive methods for more rapid changing systems, but still staying at the linear regime. In the last decade, the nonlinear properties of ANN have been used to address identification and control of non-linear systems as indicated in the introduction.

In this chapter a brief description of classical adaptive control schemes is given, and a description is also given of how the neural networks employ the classical schemes. A new method to form a neural-network adaptive controller is suggested. The idea is based on multiple-model approach together with the generalizing capability of ANN.

A. BASIC CONCEPTS OF ADAPTIVE CONTROL

The general idea in adaptive control is to find a regulator that will compensate for uncertainties, small or slowly changes in the controlled plant to form an overall stable system. There are five basic approaches to adaptive control as described by Åström and Wittenmark [AW89] including:

1) Constant gain robust control.
2) Self oscillating adaptive controller.
3) Gain scheduling.
4) Self tuning.
5) Model-reference adaptive systems.

Although the model-reference method is used in this thesis, a brief description is given of the other methods.

Robust control is based on a linear state feedback for linear MIMO systems to form a large stability margin, good disturbance attenuation and low sensitivity to small variations in the plant. This field is covered by the H_∞ and the frequency weighted LQG theories. Robust controllers designed by means of H_∞ or H_2 schemes have been applied in many areas of multidimensional systems [CS88] as well as aircraft [HG90].

The self oscillating method is also a feedback based mechanism, utilizing a relay in the feedback loop. The bandwidth is automatically adjusted to be as high as possible and to a gain margin of approximately two. However limit-cycle oscillations occur. This bang-bang type of controller is an optimal controller for some systems, in other - human operated systems, it may be unacceptable.

Gain scheduling is basically an open loop approach. The gains in a feedback loop are changed according to some measurable parameters in a pre-designed classical controller. This method was originated for flight control systems to compensate for Mach number and dynamic pressure variations.

The self-tuning regulator is based on a two-step procedure. Initially, there is an estimation process for the parameters of the plant. Based on the estimated parameters, the controller parameters are then updated under some design law. The parameter estimation is usually done recursively, using the results as if they were true. The controller design law as well as the estimation procedure vary and are problem dependent, although should tightly correspond to the plant structure.

1. Model-Reference Adaptive Control

Model-reference adaptive controllers, sometimes referred to as *model-following*, involve methods to adjust the parameters of the controller so that the overall closed loop system will behave closely to a prescribed model. It is not always possible to achieve perfect following, but the overall stability properties are determined by the reference model. Caution must be taken to ensure a stable parameter adaptation procedure. The adaptation is based on some norm of the difference between the plant output and the reference model output.

The model-reference, as well as self-tuning approaches can be further subdivided into direct and indirect adaptive controllers. In *direct control* the parameters of the controller are changed directly based on some measure of the output error, as shown in Figure 5.1. Alternatively in *indirect control* first the parameters of the unknown system are estimated, and are used to update the parameters of the controller as presented in Figure 5.2. Both direct and indirect approaches result in a non-linear controller even for LTI systems.

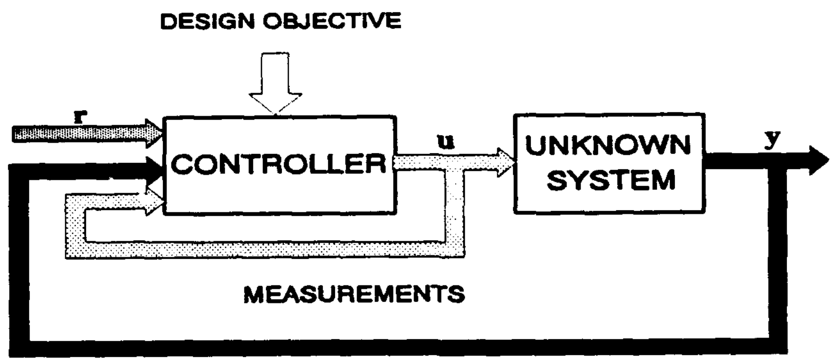


Figure 5.1: **Direct Adaptive Control**

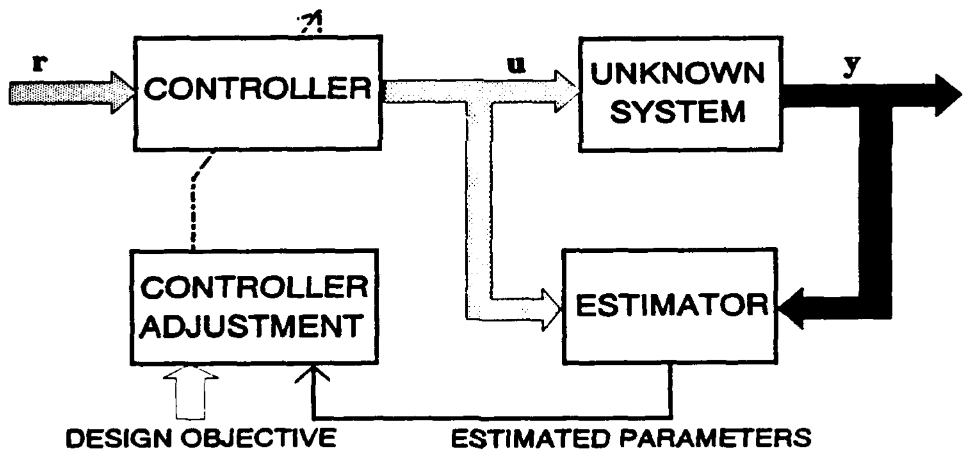


Figure 5.2: **Indirect Adaptive control**

To apply adaptive control techniques on LTI systems with unknown parameters, prior information should be available on the plant transfer function, such as the order and relative degree, and whether the zeros lie within the unit circle. The information is needed to determine the existence of the desired controller as well as the reference model. It is reasonable to assume that for non-linear system even more information might be needed for theoretical treatment. Artificial neural networks can then be of help to find a suitable controller for the non-linear systems, as presented in the following sections.

B. ADAPTIVE CONTROL USING ARTIFICIAL NEURAL NETWORKS

Artificial neural networks are used to control dynamical systems in a variety of ways. Whether it is an imitation of classical methods, or utilizing the capability of ANN to represent a non-linear relationship between input and output variables even in the case where analytical depiction does not exist. An approach such as inverse dynamics control, which is not very practical in classical methods, is quite easily done using neural networks. In any case, new ways of application are opened to facilitate control procedures that otherwise would have been very tedious, if at all possible. This section summarizes different ways of using neural networks for control.

1. Imitating Classical Methods

Many of the applications of neural networks controllers are basically an imitation of classical methods. Robust controllers derived using H_∞ method were emulated using neural networks by Bertrand to control an unstable LTI X29 aircraft model [BD91]. Neural networks were also used in the gain-scheduling approach as done by Burgin and Schnetzler for X15 pitch control [BS90], and for selecting proper gains and time-constants in a linear controller of the F/A-18 pitch controller by DiGirolamo and Donley [SM92]. Nguyen and Widrow used an ANN for the truck backer-upper problem in a self-tuning fashion [NW90]. Ungar addressed the problem of bioreactor control using neural networks in term of self-tuning regulator [MSW90, pp.387-402], and Chen directly implemented ANN for self tuning adaptive controller for non-linear SISO plant [CF90]. The most common approach is the model-reference controller, since it has the least restrictions on the knowledge of the controlled plant. Examples of model-follower neural networks controllers for non-linear dynamical systems have been given among others by Narendra and Parthasarathy [NP90], and by Saerens and Soquet [SS89] for a variety of different plants. Comparisons between neural controllers and classical adaptive controllers [SSRB90], [KC90] show that while it is worthwhile using neural networks for control, there are still areas to investigate in utilizing neural controllers in practical and theoretical issues. The model-reference approach adopted in this research is discussed later in more detail.

The method of inverse-dynamics control is a quite complex procedure as classical approach and is rarely used [IA89, pp.181-183 and 234-243]. Neural network implementation of inverse control is more natural, as discussed next.

2. Inverse-dynamics Control

Inverse-dynamics control is based on supervised learning, where the inputs to the ANN are the outputs of a system, and the desired outputs are the inputs to the system. A schematic of the procedure is given in Figure 5.3.a. When the network is trained and connected in series as in Figure 5.3.b, exciting with the desired output response, y^d, will produce the corresponding control action such that the output of the system, y, is close to the desired output value, i.e., $y \approx y^d$.

The inverse plant was used by Bertrand to control a LTI unstable X29 aircraft [BD91], and by Teo for a non-linear SISO systems [TC91]. Kawato used a slightly different scheme of inverse dynamics for control of a multi-joint arm trajectory with a procedure to update the inverse plant when the dynamics of the system changes [MSW90, pp.197-228], and [KIS88].

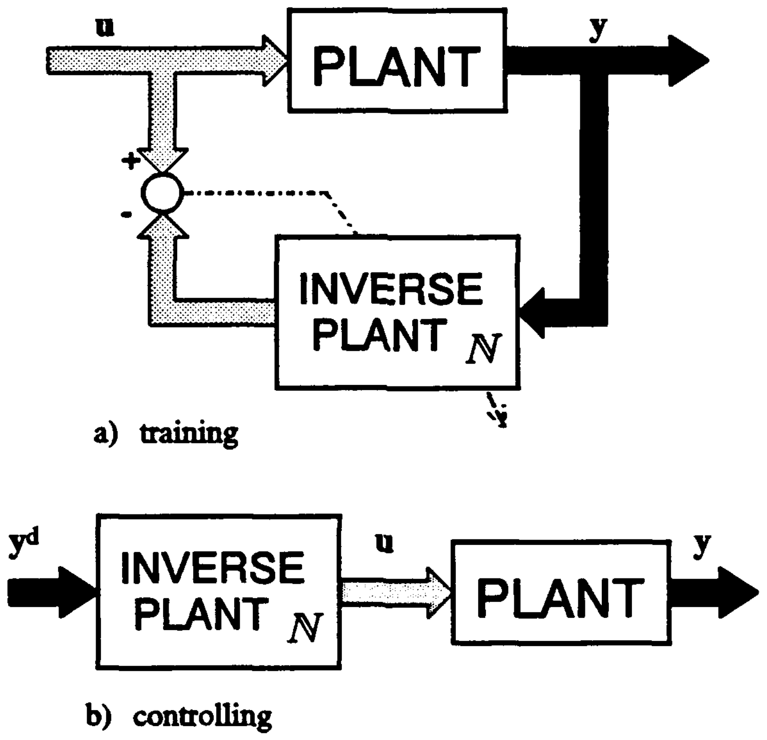


Figure 5.3: **Direct Inverse Control**

There are two major drawback of the inverse modelling. The first assumes that an inverse-plant exists, which imposes a limitation on the family of systems that can be controlled using this method. Second, it is rather sensitive to changes in the original plant, and on-line adaptation is rather long. Kawato also addressed the problem when the inverse was not unique, i.e., there is no one-to-one mapping, as when different inputs produce the same output. The non unique mapping arises also when there are more actuators than degrees of freedom. But, when such an inverse mapping does exist, it is a quite straight forward and promising procedure. Indeed the H_∞ theory often involve elements of an inverse of the plant.

3. Model-Reference Control Using Neural Networks

Model-reference adaptive control is based on reference model that specifies the behavior of the overall system. The controller parameters are updated based on the error between the outputs of the reference model, y^d, and the plant, y. Figure 5.4 shows the general architecture of model-reference adaptive controller, where, r, is the command signal fed into the reference and the controller, and u, is the control actions obtained from the controller and excite the plant. Two loops are involved. The inner loop specifies the feedback controller, whereas the outer loop adapts the controller's parameters.

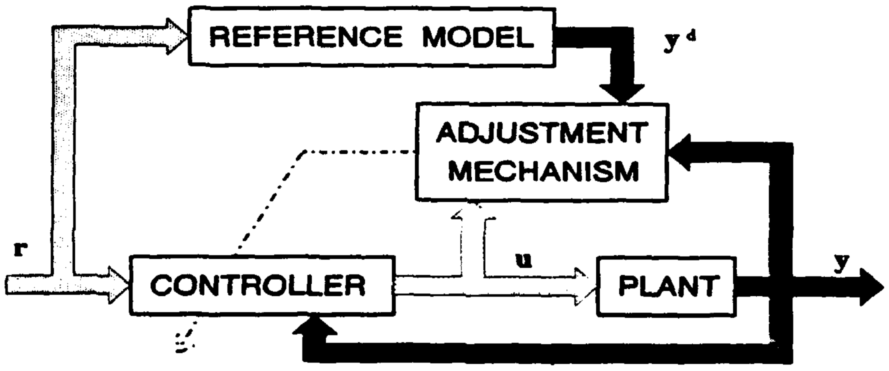


Figure 5.4: Model-Reference Adaptive Control Architecture

The model-reference approach overcomes some of the limitations of the inverse plant control. First it does not depend on the existence of an inverse plant. From the beginning of the training the control actions are in the desired trend to bring the plant to follow the reference model, i.e., there is no need for a preliminary off-line learning phase. The procedure is directed to work on the desired domain which is the output of the plant, despite the fact that the controller does not have direct access to plant output, which is usually needed for training artificial neural networks. Since it is continuously learning to supply control actions in the desired direction, it can be used for varying systems. The problem is to ensure stable learning procedure over the possible changes in the system, which is not guaranteed. The adaptation to the new plant is also a long, time-consuming process, in particular when stable learning is embraced, thus questioning the capability of real-time application. The procedure suggested in this work addresses the stability and real-time limitations as discussed in section V.C.

Model-following can be applied in neural networks both in the direct and indirect schemes. The indirect model reference control scheme as presented in Figure 5.5 involves a preceding or interlacing phase of emulation of the plant. First the emulation network is trained and the weights are frozen. At the second phase the error at the output with respect to the reference model, is back propagated via the emulated plant and through the controller network, updating the weights of the controller to drive the output of the plant to follow the reference model. This approach was used by Psaltis, Sidris and Yamamura for coordinate transformation [PSY88], and by Jordan for a six-degree-of-freedom manipulator [JI89].

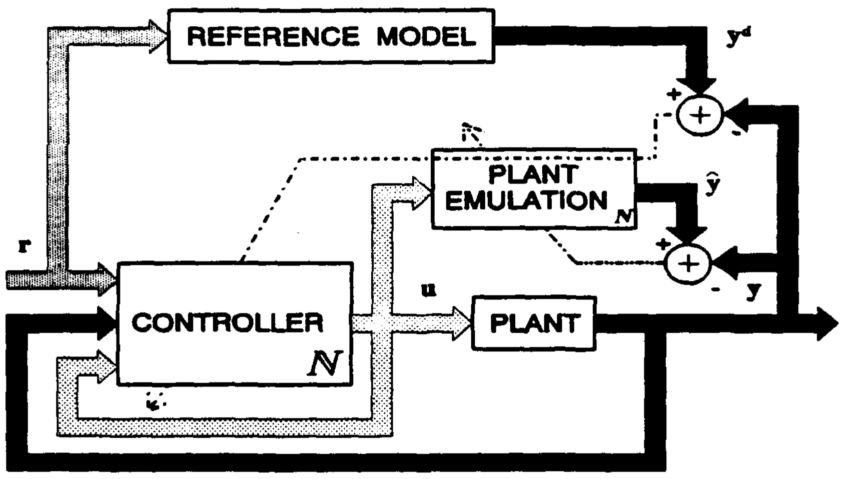


Figure 5.5: **Indirect Model-Reference Adaptive Control**

Direct model-reference adaptive control overcomes the need to first emulate the plant by utilizing the back-propagation through the plant described in Chapter III.E. A schematic of the direct model-reference controller is given in Figure 5.6. The neural network controller is trained based on an error, transformed backwards through the plant, and effects the plant output directly. This method was used by Psaltis, Sidris and Yamamura for coordinate transformation [PSY88], and by Saerens and Soquet for different constant dynamical systems [SS89]. Ha, Wei, and Bessolo supposedly used the method for control of a linearized longitudinal approximation of the F16 aircraft.

The direct model-reference adaptive control with the extensions described in section III.E is the method used in this research. The specific implementation of the control scheme is presented in the coming section.

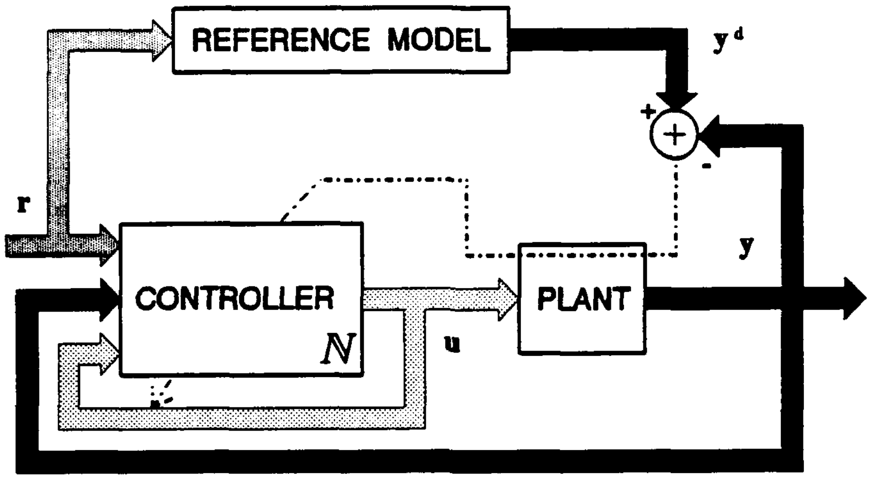


Figure 5.6: **Direct Model-Reference Adaptive Control**

C. ANN ADAPTIVE CONTROLLER FOR TIME VARYING DYNAMICAL SYSTEMS

The ultimate function desired from an adaptive controller is the ability to control a dynamical system under changing environmental conditions as well as changes in the plant itself. The dynamical systems are assumed to be output-controllable under all the changed conditions for the purpose of this design [OK90, p.706]. The adaptive control scheme is of the model-reference type. The reference model represents the desired response of the overall controlled dynamical system. The reference model is usually described by a linear dynamical system from $r \in \mathbf{R}^l$ to $y^d \in \mathbf{R}^n$, simply because the properties of linear systems are well defined. It is assumed that the reference model is chosen such that for a given bounded command signal, r, there exists a control action, $u \in \mathbf{U}$, exciting the plant, such that the output y follows y^d. The control actions spanned by $\mathbf{U} \in \mathbf{R}^m$ and are bounded in accordance and as required by the plant. This in turn makes the controller be a bounded system by itself. Under the specified control actions, $u \in \mathbf{U}$, the neural network controller treats the "non-exact" tracking, by reaching the best controller in a least-mean-square sense. The changing plant is also assumed to remain within the same class of dynamical systems and retain the same inputs and outputs, i.e., l, m and n are constant for the problem.

1. Design Philosophy

The ANN based controller design is adopted from classical control. A feedback controller $N_{fk} \in \mathbf{R}^n \rightarrow \mathbf{R}^m$ and a forward shaping filter $N_{fd} \in \mathbf{R}^l \rightarrow \mathbf{R}^m$, as shown in Figure 5.7 schematically. The two parts of the controller can be combined into one network which accomplishes both functions, having as inputs the command signal r, and the plant output y, resulting in the control actions u. The dash-dot (-··-) line in Figure 5.7 indicates the combined controller, denoted N_c. The training of the network is done as described in Figure 5.6 for the model reference approach.

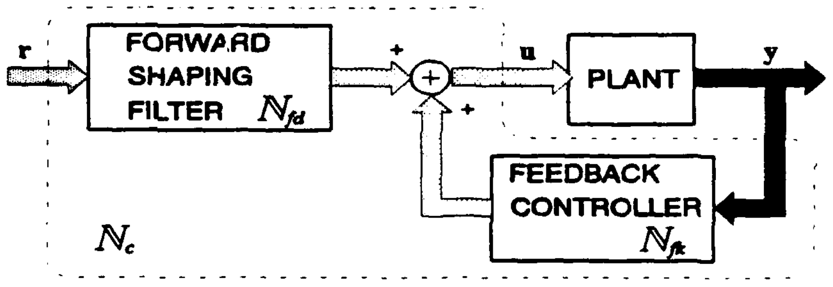


Figure 5.7: **Control Architecture**

The controller in the model-reference scheme is continuously adjusting to change the parameters in the network to drive the plant to follow the reference output.

As mentioned earlier, it is desirable to shorten the transition time of the adaptation between changes of the plant.

Training can be accomplished by defining a finite set of probable changes, each with its corresponding reference model and a resultant controller. Each model is trained off-line. As discussed in the multiple-systems emulator, a single network is used to incorporate all of the controllers creating a *unified controller*. Inputs to the controller network are used to relate the controller to the proper plant or new plant. For linear MIMO systems described in a state-space observer canonical form, controllability implies identifiability [LL87, pp.119-123]. A sensible choice of physical information to describe the plant, even in the non-linear case, is past measurements of input and output of the plant (see Figure 5.8). Having the information about the current plant in physical terms enables the network to generalize a suitable controller for the cases where the real plant differ somewhat from the plant the network was trained on. Consequently the network is valid for a variety of changes with a finite number of pre trained conditions, unless the given change is in some sense linearly independent of the models used to train the network. The structure of the unified controller is herein presented.

2. ANN Structure for the Unified Controller

The unified controller can be introduces in a network very similar in structure to the multi-system emulation network. Figure 5.8 shows the general block diagram of the training. One can see that the information on which a plant is currently controlled is passed to the reference as well, enabling a possibly different reference model for every plant. The network N_c, not only trained as a simple controller, but also makes a synthesis of type information to generate proper control actions to the corresponding plant.

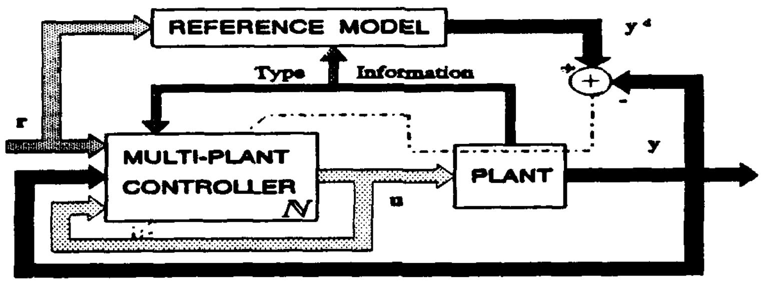


Figure 5.8: **Unified Controller Training via BPTP in a Model-Reference Scheme**

In the case of dynamical systems, the information about the *type* of the plant happens to be the same as needed for creating the controller itself. Previous measurements of the commands, output and controls are provided inside the network by using regression layers. Figure 5.9 depicts the classical controller as appears in Figure 5.7 with additional type information combined into a single network as in Figure 5.8.

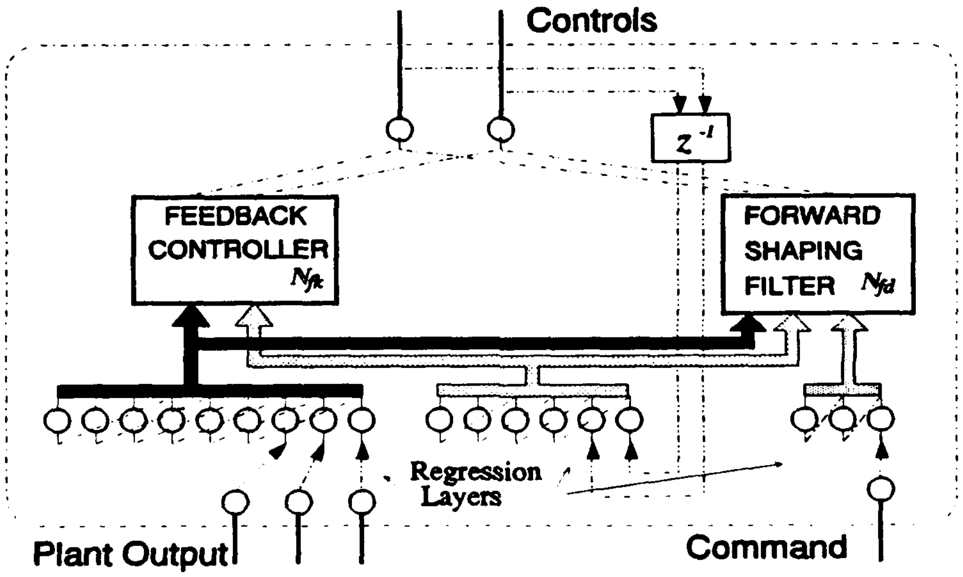


Figure 5.9: **General Structure of ANN Unified Controller**

The example given in Figure 5.9 shows a controller network with one command input, two control actions as an input to a plant with three outputs, all of which are fed into regression layers of order three. Each of the sub-networks, N_{fd} and N_{fk} in Figure 5.9, may have its own dynamics based on the inputs to that block. The blocks of the feedback controller and the forward shaping filter incorporates networks of the from appears in Figure 4.2. The type information, i.e., the regressed value of the inputs and outputs of the plant are supplied to both sub-networks. The z^{-1} block represent a delay of one time unit, and is inherent in the structure of the network. The structure of the network is further generalized in section VI.D to suit a wider range of problems and arrangements.

VI. DESIGN CONSIDERATIONS

Before starting to work with neural networks, whether for emulation or control, several design considerations need to be taken into consideration. There should be some general knowledge about the system such as its interface with the environment. The following issues need to be addressed:

1) Defining measurable input and output variables.

2) Frequency and amplitude ranges of interest.

3) Sampling rate.

4) Exciting input character.

5) Data record length.

6) The class to which the dynamic system belongs, if possible.

7) Normalization and scaling.

8) Capabilities of the peripheral equipment on hand, such as measurement devices, analyzing tools - hardware and software, etc.

According to the available information, a suitable network is constructed and properly excited. To verify that indeed the network has been trained to meet the design goal in the domain of interest, the following matters must be looked into:

a) Selecting a design criteria.

b) Define and execute validation issues.

The above parameters can be called *design variables*, and the manner by which they influence the design of a neural network is discussed in this chapter.

A. DYNAMICAL SYSTEM PROPERTIES

In order to have a successfully trained network, the dynamical system has to have certain properties, depending on the task performed, i.e., emulation or control. The properties were defined earlier and are repeated here to make the discussion about design issues more comprehensive. The systems under consideration in this research are nonlinear MIMO dynamical systems. For both emulation and control the systems are assumed to be bounded, not necessarily BIBO stable (see IV.B).

1. System Properties for Emulation Design

In order to successfully emulate a system, or multiple systems in a neural network the system has to be reachable in the linear case, or realizable for nonlinear systems (see IV.B). The measured variables are the exciting inputs and the plant outputs.

2. System Properties for Controller Design

In order to successfully control a system, or multiple systems, in a neural network the system has to be output-controllable (see V.B). For controller design the measurable variables are the command signals, the control actions and the plant outputs.

B. SPECTRUM AND AMPLITUDE ISSUES

In analyzing dynamical systems the frequency domain of interest has to be defined. For emulation, the spectrum is usually defined by the dynamical behavior of the plant. In control applications, the bandwidth of the reference model, or the spectrum over which the controlled system has to function, determines the frequency range. For nonlinear systems it is important to define the amplitudes of the measured variables both for emulation and control problems. The trained network will perform well within the specified amplitude range, but there is no guarantee on the performance outside this range. In addition to the above conditions, the sampling rate, exciting signal characteristics complying with the concept of persistency of excitation, both in frequency and amplitude, and consequently the amount of data collected need to be considered, and are discussed in Chapter VIII while examining the test case.

It is of great importance in neural networks that the inputs to the network have similar energy content over time for proper training of the weights. Since a network is also a non-linear device, matters of amplitude should be considered for proper operation in the test phase. Therefore, the input and desired output values are scaled and biased from the physical values to fall within the acceptable range of the network, usually [0,1] or [-1,1]. If the energy content of certain inputs differ from others, say greater, it should be further compensated by the scaling or by selecting a smaller learning rate, to let the input equally effect the network. The relative significance will eventually be seen by the final values of the weights. Failing to correctly scale and normalize the network in its interface with the environment, may prohibit successful training within a reasonable long training period, if at all.

C. NETWORK STRUCTURE SELECTION

The specific structure of ANN per problem is generally the choice of the designer. This selection is not unique as will be shown below. The structural constraints and options are discussed in this section.

One constraint is defined by the interface with the environment which specifies the number of input and output nodes in the network. The inner structure is more problem dependent. As explained in Chapter IV, it is desirable to have the inner structure similar to the class of model to which the system is suspected to belong. The resemblance gives a better insight to the system when analyzing the resultant network, i.e., there is more to a network than just a mesh. In some cases it can lead to almost perfect representation with full physical interpretation of the weights, as demonstrated with the LTI systems (IV.C). In reality noise is present in the measurements, as well as in the process, in which case the "perfect" is in the mean-square sense.

The similarity between the emulation and control network structures, led to the formation of a general structure, suitable to generalize the procedure for emulation, control and related processes.

A general ANN structure is introduced in this section. The structure embodies the general interface needed for emulation, identification, classification or control of dynamical systems, given in state-space or time series form. The general structure is presented in Figure 6.1.

The input layer may consist of any or all of the following parameters:

- Forced excitation - used to drive the dynamical system with the suitable inputs, discussed in subsection B.
- Unforced excitation - other inputs to the network, which are not the direct excitation, but may result from the forced excitation. For example, an output of a reference model.
- Type - information used to indicate a certain system. Used especially in multiple-system networks.

Any portion or combination of the input layer may be passed to the variable-order input-regression layer as shown in Figure 6.1.

The output layer may consist of any or all of the following parameters:

- Dynamic output - used for varying outputs, such as the output of a dynamical system.
- Static output - used e.g., for classification, or other non-varying output parameters.

Any portion or combination of the output layer may be passed to the variable-order output-regression layer.

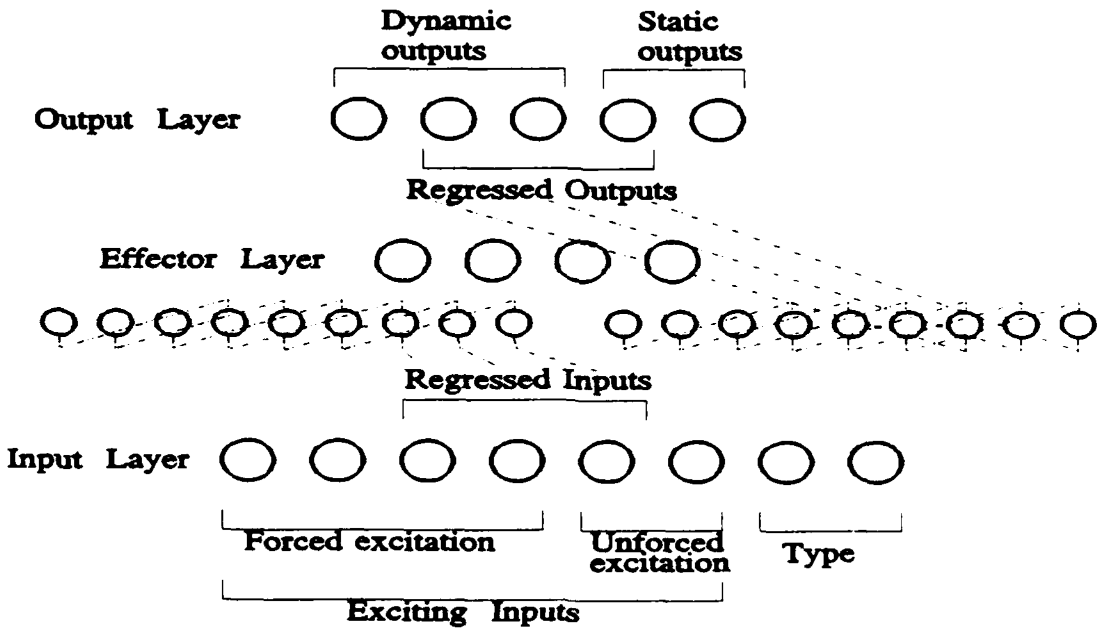


Figure 6.1: A General Structure for Multi-Layer ANN

The effector layer seen in Figure 6.1 is introduced if some intermediate values were needed to describe a more complex system. For example, combining two networks where one feeds the second, as in control problems. This layer is also regressible through the output regression or via the unforced excitation.

Different structural combinations are possible to solve the same problem. The inner structure, as discussed earlier is the choice of the designer, and problem dependant.

A program to interface with NeuralWare's software was written in C, based on this general framework for a multi-layer ANN, with easy accessible user-defined functions to accommodate a variety of systems, structural combinations and excitations. This USERIO program appears in Appendix D, along with a dedicated control strategy that monitors the execution of the USERIO program.

This section established this structure of artificial neural networks in conjunction with non-linear dynamical systems. A new method is now proposed to define minimal realization of the inner structure.

D. MINIMAL REALIZATION OF A MULTI-LAYER ANN

Multi-layer networks consist of an input layer, an output layer, and at least one hidden layer. A typical feed-forward network with two hidden layers is presented in Figure 2.2. The layers are fully connected, and the connections form a weight matrix *W*. Theoretically, there is no limit on the number of hidden layers or the order of layer connection, i.e., the connected layer does not have to be consecutive. A bias term is usually added to each PE with an associated non-zero weight that can be considered as a threshold of that PE. The number of inputs and outputs is dictated by the nature of the problem which the network has to solve or represent. The number of hidden layers, their arrangements, the number of PEs in each layer, and their connections are the designer's choice.

Separate theorems express the existence of artificial neural networks as mapping devices from a finite dimensional space to another to any desired degree of accuracy with only one hidden layer, as discussed in section II.E. Kolmogorov's theorem [HN90, pp. 122-124] gives an upper limit to the number of hidden nodes needed for the mapping ($2n+1$, where n is the number of input nodes). However, for every mapping task a *different unknown* activation function is needed. It has also been shown [HSW89], that ANN are general approximators of a bounded function using an *arbitrary squashing* activation function, i.e., sigmoid or hyperbolic tangent, in an *unknown but large enough* number of PEs to get the desired accuracy. However, neither theorem really helps to build a "correct" structure of the network.

An understanding of the physical nature of the problem may assist in choosing a "good" number of hidden layers and hidden nodes. Until the present, choosing network parameters was a kind of art usually done by trial and error. A more rigorous way of selecting the number of hidden layers and how many nodes each layer should include is presented in this dissertation.

Although the method cannot predict in advance a good choice for the number of layers and PEs, the rationale for going through the procedure of finding minimal realization of the network is to achieve the same accuracy with less network complexity. It is especially useful in recall mode by having less computation and less parameters in the network, which leads to easier implementation and faster reaction. In training mode however, there is no distinctive conclusion on total time required for complete training. For off-line training there is no major impact but it may be useful in on-line processes if it can be found that a much simpler network, which trains faster, can still do the job. Tests showed that usually the minimized network needed more iterations to converge but the overall training time has to be examined per task, and might even be shorter since less computation per iteration are involved.

1. Selection of Number of PEs in Hidden Layers

The method developed finds a minimal realization of the network, by means of number of hidden layers and number of PEs in each layer. It is assumed that there exist a network, trained to the desired accuracy, containing a large enough number of layers and PEs. The assumption is based on the results of Hornik, Stinchcombe and White [HSW89]. Thus, the network is assumed to have an adequate number of hidden layers, each containing large enough number of PEs, where every PE incorporates a squashing activation function. The idea is to find how many PEs carry most of the activation level from one layer to the next. It is done by examining the singular values of the weight matrix connecting the two layers.

The singular values of a rank q matrix $W \in \mathbf{R}^{m \times n}$, denoted by σ_i, are the non-negative square root of the eigenvalues of $W^T W$ ordered such that $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$. If $q<n$, $(n\text{-}r)$ singular values are zero, i.e., $\sigma_{q+1}=\sigma_{q+2}=\ldots\sigma_n=0$. There exist two unitary matrices $U \in \mathbf{R}^{m \times m}$, $V \in \mathbf{R}^{n \times n}$ and a diagonal matrix $\Sigma \in \mathbf{R}^{m \times n}$ such that,

$$W = U \Sigma V^T = \begin{bmatrix} \Sigma_q & 0 \\ 0 & 0 \end{bmatrix} \tag{6.1}$$

where $\Sigma_q = diag(\sigma_1, \sigma_2, \ldots \sigma_q)$. This partition is called the singular value decomposition (SVD) of the matrix $\boldsymbol{W}$. The greatest singular value σ_{max} is given by σ_1.

Since the weight matrix contains the information on how the data is passed from source layer to the target, the singular values of the matrix are indicative of the characteristics of the matrix and hence serve as a feature detection process. The specific features embedded in the weight matrix need to be extracted. To identify the features, a trace of the singular values of the weight matrix is calculated while training progress.

Figure 6.2 shows a typical trace of the singular values of the weight matrix as a function of training iterations, where the unreduced target layer has seven PEs. The traces of each singular value is denoted by S1,S2,...S7. The singular values carrying the important information will develop and can be identified from the rest that do not carry significant information. In a multi-layer network , the weights are usually given a random initialization within a specified range, carrying values other than zero. To be able to identify even very small but significant feature, the initial randomization should be small enough to let the small singular values emerge above the initial values. The trace marked by S6 is an example of a significant singular value that pops-up over singular values that initially were bigger but do not carry significant information.

It should be noted that the singular values are traced continuously, and not necessarily sorted in descending order as defined by eq. (6.1). ANN that are trained using the back-propagation algorithm must have initial values other than zero. Small initial values are theoretically acceptable but may cause difficulties at starting the training process.

Based on the singular values an approach to select the number of PEs needed, is to define a threshold λ, above which the contribution to the inputs to the target layer is considered significant, and set the number of PEs in the target layer as the number of singular values above the threshold. The threshold is usually related to σ_{max} by a *variation ratio* - ν as,

$$n_{PEs} = \max_{i=1,2,\ldots,n} \{i \mid \sigma_i \geq \lambda\} \tag{6.2}$$

$$\lambda = \frac{\sigma_{max}}{\nu} \tag{6.3}$$

In the example given in Figure 6.2, the estimated number of PEs needed is four out of seven initially set.

However, using the singular value decomposition to determine the number of PEs needed in the target layer, implies that initially the number of PEs in the source layer must be greater or equal to the number of PEs at the target layer.

The plot of singular values can also serve as a useful tool to determine if the training is completed, by noticing that the singular values have reached their steady state values, and more training will not give any improvement.

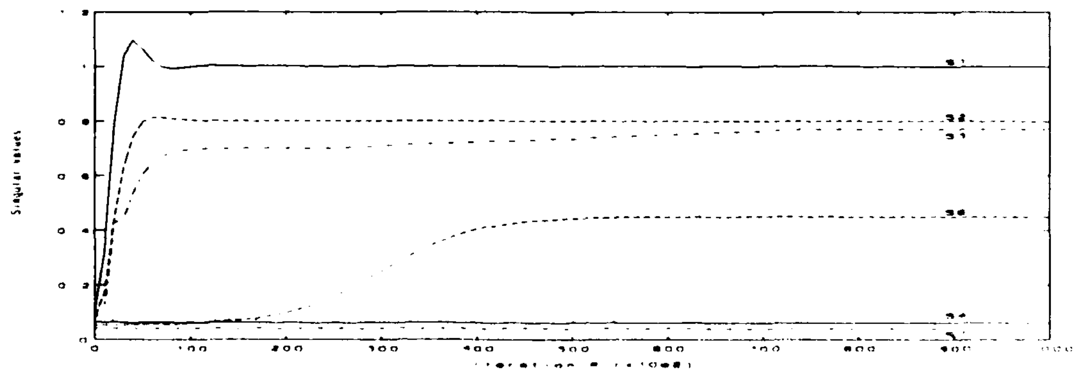


Figure 6.2: Singular Values of a Weight Matrix as a Function of Training Iteration

2. Selection of Number of Hidden Layers

The number of hidden layers needed is determined experimentally in the following way. It is assumed that the network was previously constructed and successfully trained with an adequate number of layers. A layer is said to be redundant when the SVD plots of consecutive layers show the same number of significant singular values, and their traces are identical up to a scale factor.

3. Selection of Regression Order

The order of the regression layer, i.e., the number of previous measurements taken into account as an input to the network, should first be based on physical or theoretical reasoning. If no information on the order were given, the initial regression layer would have to be of large enough order assumed to be sufficient, e.g., estimated order of the system. The number of input and output histories is determined by observing the weight matrix connecting the regression layer to the hidden layer. Figure 6.3 shows a trained weight matrix from a regression layer of order eight, with three regressed values, to a hidden layer with five PEs. The height of each box is the value of the specific weight, and the same shading pattern refer to the same regressed variable. From Figure 6.3 it is seen that the contribution of regressed values prior to *t-4* are negligible, and thus can be eliminated leaving the regression layer of order four only.

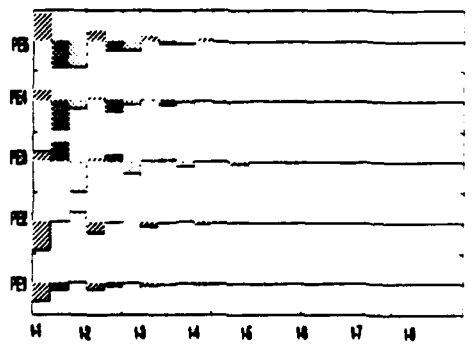


Figure 6.3: **A Weight Matrix From Regression Layer to Hidden Layer**

E. VALIDATION ISSUES

To conclude the design considerations, one must define a design criterion, and supply the means to verify that indeed the goal has been achieved. It is not sufficient to require that some norm of the output error, for either emulation or control, will meet some threshold. It has to be proven that the task is fulfilled over the whole domain of interest. The concept of consistency of excitation holds here as well as some other measures to verify the results.

In dynamical systems, a common way is to check the response in the time domain, to some known test cases and analyze them. Common time-domain tests are pulse, step, ramp or dipole commands, as well as checking the performance in presence of noise and evaluating the results statistically. A complimentary evaluation is done to verify the spectral properties. A frequency response is calculated in parametric or non-parametric means, and compared to previous knowledge on the system. The location of poles and zeros can be extracted, to give more physical interpretation on the system.

With neural networks, another means to evaluate the results is done by examining the weight matrices of the trained network, assuming that there is some correspondence between the network structure and the physical behavior of the system. An example is given in the minimal realization procedure (VI.C), and by the LTI system emulation (see IV.C). For example, from the weight matrices one can estimate the number of previous measurements needed to represent the system, which gives information on the order of corresponding ARMAX model.

Most of the above criteria and techniques are used in this research to verify the results, as demonstrated in Chapter VIII.

Up until now, the theory behind neural networks and training mechanisms was established. It was shown how ANN can be utilized to emulate and control non-linear MIMO dynamical systems. Ways to construct a suitable network, train and evaluate the results were discussed. It is now appropriate to demonstrate all the above in an example, and show the strength and capabilities of artificial neural networks in emulation and adaptive control. The rest of the dissertation involves the presentation and the evaluation of a representative test case.

VII. ANN EMULATION AND CONTROL DEMONSTRATION

This chapter presents a test case to examine implementation of emulation and control using artificial neural networks as developed in the previous chapters.

The F/A-18A fighter aircraft was chosen as a demonstration MIMO dynamical system due to the availability of multiple control surfaces to a rather co..iplex dynamics, see Figure 7.1 below. A damaged aircraft was assumed to represent a change in the system, which the neural network emulates and controls. Thus, the neural network forms a reconfigurable adaptive controller for an impaired aircraft. The closed-loop performance is dictated by a reference model that meets the conditions required by military specification MIL-F-8785C. The discussion continues with a description of the open-loop dynamic model of the F/A-18 followed by specific damage mechanism which were treated in this research.

A. F/A-18A LONGITUDINAL DYNAMICAL MODEL

The F/A-18A linearized longitudinal dynamics with asymmetrically-saturated control inputs, provides a 25th order dynamical system with six control inputs and three output variables. The dynamical model includes the airframe dynamics as well as actuator and sensor dynamics. The continuous state-space model was developed from data in [MDC82] for a flight condition of Mach 0.6 and altitude 10000 feet, in escort configuration. The aircraft equations of motion are taken from [MAG90, pp. 256-260, 296-298] in stability axes and trim conditions, assuming no gusts. The actuator and sensor models are based on the model developed by Rojek [RF86].

Figure 7.1 shows the F/A-18 with the control surfaces and describes the sign conventions for both deflections and aerodynamic coefficients. The terms in body axes are denoted $(\bullet)_B$ and $(\bullet)_S$ denote stability axes. The control surfaces affecting the longitudinal dynamics are the stabilators (dst), leading edge flaps (dle) and trailing edge flaps (dte).

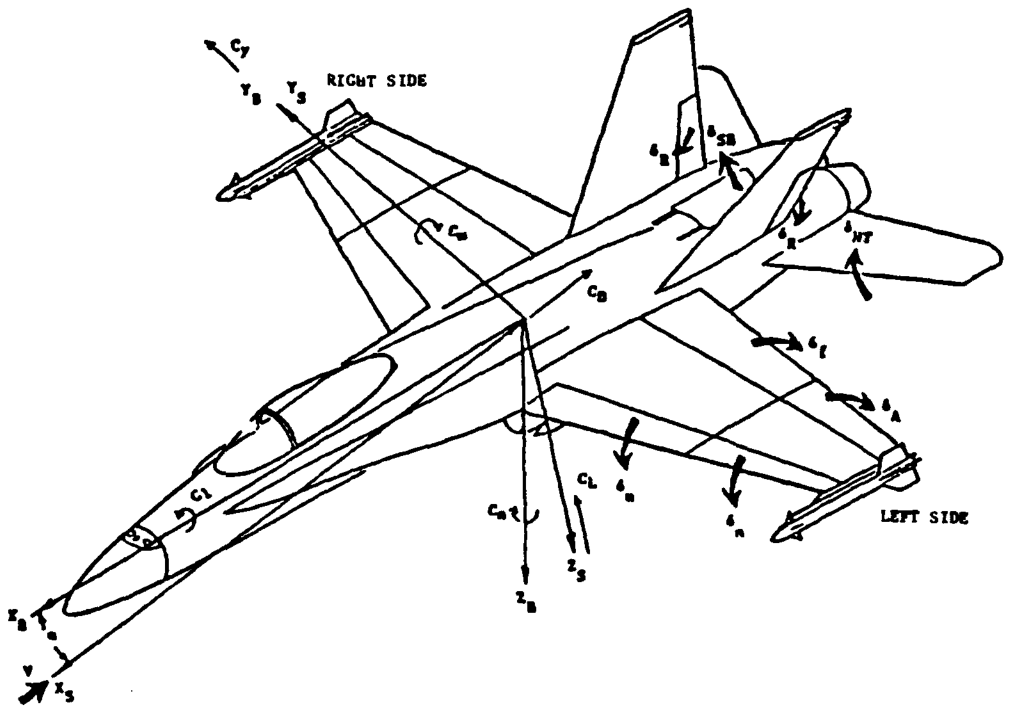


Figure 7.1: **F/A-18A Axes, Control Surfaces and Sign Conventions**

The continuous state-space representation of the aircraft, which includes the sixteenth order actuator dynamics, fourth order airframe dynamics, and fifth order sensor dynamics is given in the 25th order augmented system as follows.

$$\begin{cases} \dot{x} = Fx + Gu \\ y = Hx + Du \end{cases} \tag{7.1}$$

The control vector, u, is the actuator deflection commands [deg],

$$u = \left\{\delta_{st_r}\ \delta_{st_l}\ \delta_{le_r}\ \delta_{le_l}\ \delta_{te_r}\ \delta_{te_l}\right\}^T \tag{7.2}$$

where *st, le and te* denote the stabilators, leading edge and trailing edge, and the subscripts *r,l* refer to the right and left sides respectively. The deflections of the surfaces are limited to:

- Stabilators [+10.5°,-24°]
- Leading edge flaps [+34° ,-3°]
- Trailing edge flaps [+45° ,-8°]

The measurable output, y, consists of,

$$y = \{q\ n_z\ \alpha\}^T \tag{7.3}$$

where q is the pitch rate [deg/sec], the normal acceleration n_z [g's] and α is the angle of attack [deg].

The numerical values of the aerodynamic stability derivatives together with the actuator and sensor dynamics appear in a MATLAB™ script file, *f18longN.m*, that calculates the augmented system, and are given in Appendix A, with the eigenvalues and the corresponding damping and frequencies appearing in Table 7.1. The program also calculates the matrices for the impaired model described next.

Table I EIGENVALUES, FREQUENCIES AND DAMPING RATIOS OF UNDAMAGED F/A-18A LONGITUDINAL CONTINUOUS MODEL

S-Plane roots	Damping	Frequency [rad/sec]	Mode
-0.0054 + 0.0569*i*	**0.095**	**0.057**	**Phugoid**
-0.0054 − 0.0569*i*	**0.095**	**0.057**	**Phugoid**
-0.9786 + 2.6341*i*	**0.348**	**2.810**	**Short Period**
-0.9786 − 2.6341*i*	**0.348**	**2.810**	**Short Period**
-14.000	1.000	14.00	AoA Sensor
-14.924 + 33.200*i*	0.410	36.40	Stab. Act. right
-14.924 − 33.200*i*	0.410	36.40	Stab. Act. right
-14.924 + 33.200*i*	0.410	36.40	Stab. Act. left
-14.924 − 33.200*i*	0.410	36.40	Stab. Act. left
-24.850 + 24.647*i*	0.710	35.00	TE Flap Act. right
-24.850 − 24.647*i*	0.710	35.00	TE Flap Act. right
-24.850 + 24.647*i*	0.710	35.00	TE Flap Act. left
-24.850 − 24.647*i*	0.710	35.00	TE Flap Act. left
-26.900	1.000	26.90	LE Flap Act. right
-26.900	1.000	26.90	LE Flap Act. left
-62.127 + 85.020*i*	0.590	105.3	Stab. Act. right
-62.127 − 85.020*i*	0.590	105.3	Stab. Act. right
-62.127 + 85.020*i*	0.590	105.3	Stab. Act. left
-62.127 − 85.020*i*	0.590	105.3	Stab. Act. left
-82.900	1.000	82.90	LE Flap Act. right
-82.900	1.000	82.90	LE Flap Act. left
-167.80	1.000	167.8	Pitch Rate Gyro Sensor
-379.49 + 110.68*i*	0.960	395.3	Normal Accel. Sensor
-379.49 − 110.68*i*	0.960	395.3	Normal Accel. Sensor
-461.70	1.000	461.7	Pitch Rate Gyro Sensor

B. DAMAGED F/A-18A MODEL

The damage mechanism considered in this example is a malfunction in the actuator of the left stabilator. The left stabilator does not respond to actuation commands and remains fixed at trim condition. This is modeled by nullifying the corresponding actuator dynamics in the actuators model, as shown in Appendix A.

C. CONTROL REFERENCE MODEL

The reference model for the model-following controller is based on the military specification for flying qualities of piloted airplanes, MIL-F-8785C. The general configurations is shown in Figure 7.2. The same reference model is selected for both damaged and undamaged F/A-18 models.

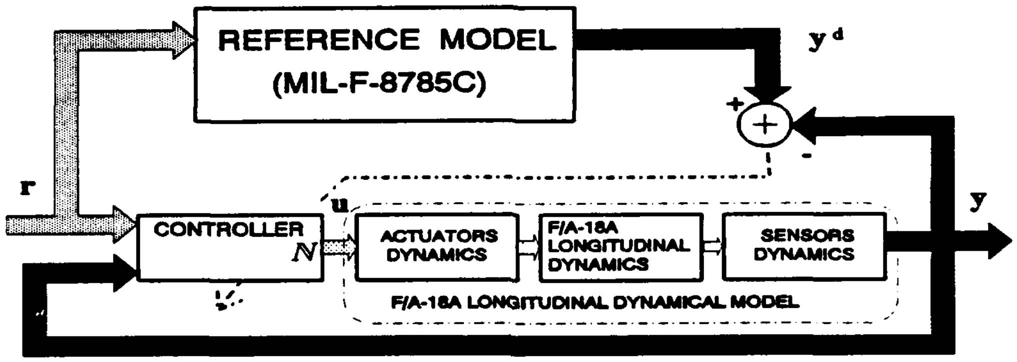


Figure 7.2: **Model-Reference Control Architecture for the F/A-18A**

The reference is a forth order linear model, that defines desirable behavior for the short period and phugoid modes [MS80, pp.12-16]. The frequency and damping are

- Short period - $\omega_{sp}=4.3$ [rad/sec] ; $\zeta_{sp}=0.6$
- Phugoid - $\omega_{ph}=0.02$ [rad/sec] ; $\zeta_{ph}=0.4$

The reference model is given in discrete state-space form as

$$\begin{cases} z(k+1) = Fz(k) + Gr(k) \\ y^d(k) = Hz(k) + Dr(k) \end{cases} \tag{7.4}$$

The input command, r, is the pilot pitch stick commands p_x. The output vector, y^d, consists of,

$$y^d = \left\{q^d \; n_z^d \; \alpha^d\right\}^T \tag{7.5}$$

where q^d is the reference pitch rate [deg/sec], the desired normal acceleration n_z^d [g's] and α^d is the reference angle of attack [deg]. The reference model is scaled such that at full range pilot command $p_x = 1$ generates a normal acceleration of $n_z^d = 5g$ in magnitude.

D. DESIGN PARAMETERS AND CONDITIONS

This section gives the test case specific parameters and conditions for topics discussed in Chapter VI as design considerations. The fact that the basic aircraft model is linear makes it easier to apply the conditions set in Chapter VI, but neural networks have been shown to successfully work with non-linear functions as well.

1. Plant Properties

Both the undamaged and damaged aircraft models are bounded MIMO systems. The models are reachable as required for emulation, and *output-controllable* as required for control application.

2. Spectrum and Amplitude Ranges

The frequency band of interest is taken to be the same for emulation and control. It is chosen to cover the open-loop and closed-loop aerodynamic modes. The poles of the actuators and sensors are much faster and left out of this range. The frequency band of interest covers three decades from f_{low}=0.005 Hz to f_{high}=5 Hz. The sampling rate in this work is 50 Hz, which lets the Nyquist frequency be five times larger than the highest frequency of interest, and about ten times the short period mode and therefore aliasing is greatly suppressed. This sampling rate causes the discretized plant to become non-minimal phase. Non-minimum zeros are accounted for in the control phase by resetting the simulation, to prevent modelling errors from growing unbounded.

The exciting signal follows the concept of *persistence of excitation*. A *random binary sequence* (RBS) is used, shaped such that the energy content of the signal will be distributed across the desired spectrum. The exciting signal coincide with the sampling rate. Since the network is non-linear the exciting signal should cover the whole amplitude range of interest. The RBS signal was found suitable for persistently exciting the network range, and was set to span [-1,1].

The characteristics of the RBS signal are shown in Figure 7.3 in the time domain. Figure 7.4 shows the spectral characteristics of the RBS signal used for training (order of 500k presentations). The drop in the spectral energy above five Hertz was intended to limit the excitation to the desired range and prevent high frequency corruption. For testing the trained network it was impractical to use such long sequences, and the shortest sequence (16384) capable of spanning the desired range of [0.005,5] Hz was used. The spectrum of a testing sequence appears in Figure 7.5, which is less uniform then the training spectrum. It should be noted that it is very difficult to properly excite the very low end of the frequency range (f=0.005 Hz is a cycle time of 200 sec. which is 10000 samples), therefore some mismatch might be expected in the low range due to insufficient excitation from practical reasons.

The amplitude bound brings up the issue of scaling. The desired range of signals for a successful learning of the network is [-1,1]. Therefore the models of the plants and the reference model were scaled and biased from the physical range to the network range. The biasing eliminated the non-linearity in the actuator's deflection range. The non-linearity of the network is expressed in multiple models handled in a single network.

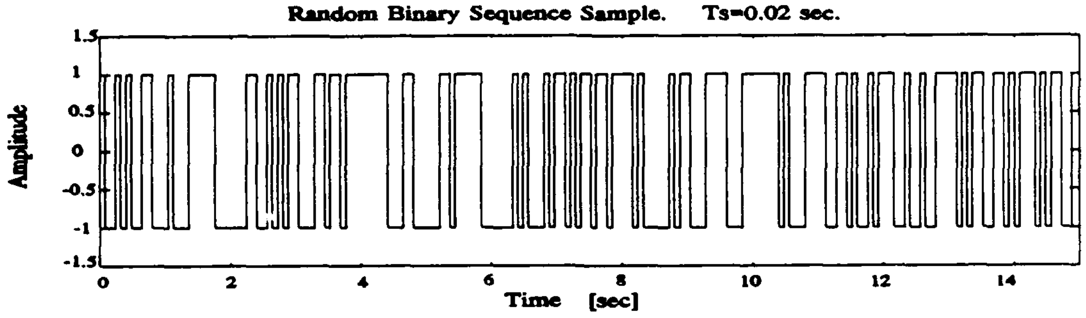


Figure 7.3: Time Domain Sample of Random Binary Sequence

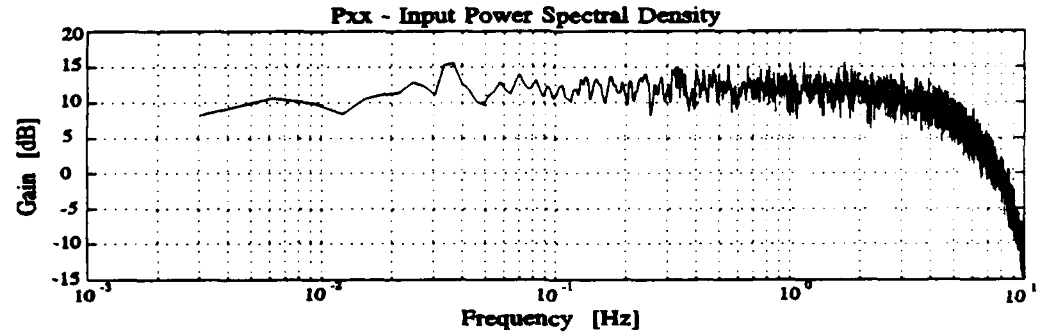


Figure 7.4: Power Spectrum of Training RBS Signal (512k presentations)

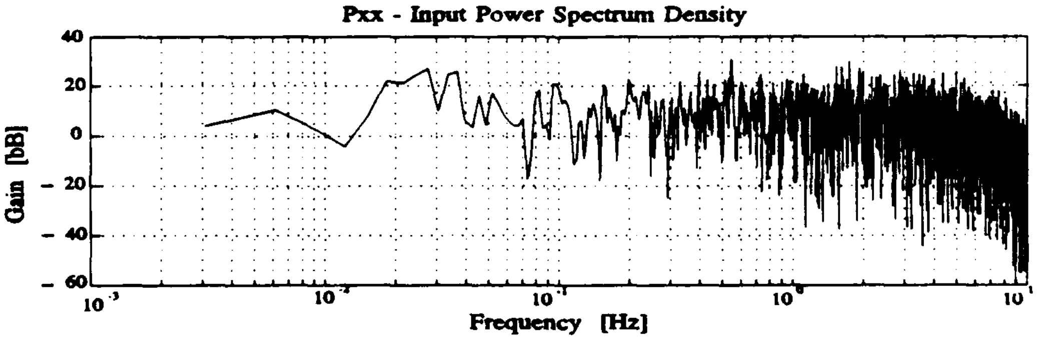


Figure 7.5: Power Spectrum of Test RBS Signal (16k presentations)

3. Network Structure for Emulation and Control of the F/A-18 Model

The multiple-plant model belongs to the class of models which are represented by model III as of section IV.A. The corresponding structure was selected as appears in Figures 4.2 and 5.9 with the appropriate number of inputs and outputs, as will be seen in the result chapter next.

VIII. SIMULATIONS, RESULTS AND ANALYSIS

This chapter describes the different experiments done with the aircraft models described in Chapter VII. The results are shown and analyzed, both in the frequency and time domains. The spectral plots of the neural network response and the desired output, used FFT analysis of the input and output sequences. The data was windowed using Hanning window. The system Bode plot is the analytic spectrum of the simulated system, which is used for generating the desired output. The neural network results are given for every relation between the input and output of the MIMO dynamical system, and compared with the theoretical and desired values.

In the following plots, *Plant #1* refers to the undamaged aircraft model, whereas *Plant #2* refers to the damaged left stabilator actuator impaired aircraft model. The experiments start with the emulation process and are followed by the adaptive controller.

The emulation results includes three parts. First the undamaged and damaged aircraft models are emulated separately in the proposed network structure that contains feedforward and recurrent sub-networks, corresponding to model III (see section IV.A and Figure 4.2). In the second part, a single network is used to emulate both aircraft models using a random balanced presentation procedure as described in section IV.D. The multiple-plant emulation in a single network is repeated with a minimized network. The minimal realization is done via the singular value analysis of the weight matrices and evaluating the minimal regression order by analyzing the weights from the regression layer to the first hidden layer as described in section VI.D.

Since the F/A-18 aircraft used in the analysis is a MIMO system with six inputs and three outputs, many plots are needed to completely describe the behavior of the system. Therefore the results shown in the following A, B and C sections are for representative variables, and the rest of the results, when appropriate, are gathered in Appendix B. The representative inputs are the right and left stabilators, noted in the graphs as dst_r and dst_l, respectively. These inputs were selected to emphasize the differences between the damaged and undamaged plants. Recall that the damage is a malfunction in the left stabilator actuator. The output variable selected is the normal acceleration n_z. Note: the results due to the selected variables are not necessarily the best among the rest of the inputs and outputs.

A. EMULATION OF A SINGLE MIMO DYNAMICAL SYSTEM

The experiment started with an initial network that had regression layers of 25th order (same order as the plant, as might theoretically be required). Examining the weight matrix from the regression layer to the first hidden layer showed that the three most recent regressions carried most of the activation and the weights of prior measurements were rather small compared to the first three. Based on this observation the regression layers were reduced to be of order three, as in Figure 8.1 for the multi-plant emulation structure, and is the network used for the results presented here. The order reduction of the regression layer greatly reduced the complexity of the network and the training time, with no loss in the accuracy of the training. This was the first reduction of the network structure, but the hidden layers were not minimized.

The input regression layer denoted as *ComReg*, is a third order regression of the six control inputs containing 18 nodes. The input regression layer is fully connected to the first hidden layer *comh1* which is further fully connected to the second input hidden layer *comh2*, each containing 36 PEs. The second hidden layer is fully connected to an output-buffer layer, *OutBuf*, through a connecting *Layer* with unit-valued fixed corresponding weights, for visual separation of forward portion of the network from the recurrent portions of the network. In Figure 8.1, the connections between layers are symbolically illustrated by the connection from a single PE at a source layer fully connected to the target layer. Showing the full connectivity will make the drawing too complex to visualize.

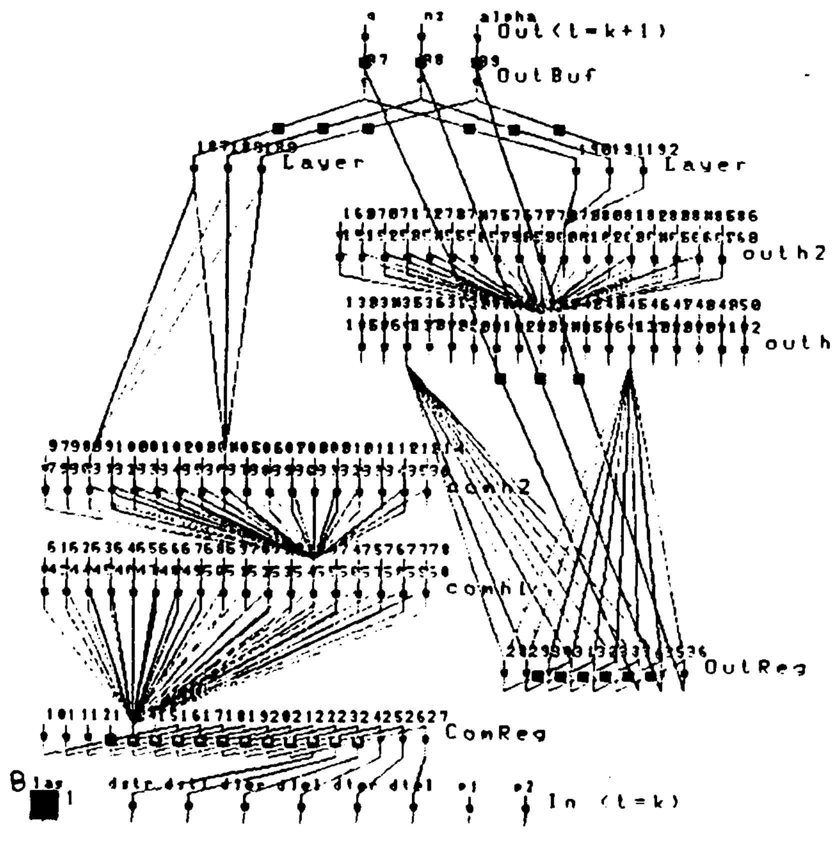


Figure 8.1: Network Structure for a Multiple-Plant F/A-18 Emulation

The recurrent sub-network starts with a third order output regression layer *OutReg* which obtains the inputs via a feedback from the output layer. The use of the output-buffer layer was required due to limitation of the current software from NeuralWare that was used in this research. The *OutReg* layer consists of nine PEs and is fully connected to the first hidden layer of the recurrent portion *outh1*, which is further connected to the second hidden layer *outh2*, each containing 36 PEs. The second hidden layer is connected to the output through *Layer*, for better visualization. In the recurrent sub-network connections are also shown only schematically, since all the layers are fully connected.

In this section each plant, the undamaged and damaged aircraft are emulated separately on two different networks structured as in Figure 8.1.

1. Single Plant Emulation - Frequency Domain Results and discussion

The examination starts with spectral analysis of the network compared to the desired model. Figure 8.2 shows the spectral response of the pitch rate q, the normal acceleration n_z and the angle of attack *AoA* of the undamaged aircraft due to the right stabilator dst_r (upper plots) and the left stabilator dst_l (lower plots). Each graph shows the analytic frequency response in the solid line, which is the Bode plot of the outputs due to the corresponding input, together with the desired outputs in the dashed line, and network output values in the dash-dot line, analyzed using FFT, while exciting the input with random binary sequence (RBS). Analogous plots for the other inputs to the system, i.e., the leading edge and trailing edge control surfaces, appear in Appendix B, Figures B.1 and B.2 correspondingly.

The figures show almost perfect match between the system Bode plot, the desired output and network output frequency response over the range of 0.03 Hz to 5 Hz. The deviation between the network output and the desired output is mainly a problem of persistent excitation. Two factors contribute to that difference. First it is very difficult to properly excite at such low frequencies (0.01 Hz is a cycle time of 100 seconds and in 5000 learning iterations only one change is expected). The other factor is the inherent fact that as the frequency goes down, the number of presentations of data information due to that frequency goes down. The less a feature is presented to a network, the less trained the network is to correctly respond to similar excitation.

The deviation of the network output and the desired output from the analytical Bode plot is due to the coarse resolution at the low frequency. Since the spectral responses of the network and the desired output are done using FFT analysis, the spectrum is linearly spread. In order to get better resolution over a range of three decades, the amount of data needed to be collected becomes prohibitive with respect to the capabilities of the current analyzing tools (software and computer memory). Due to the reasons mentioned above and the random nature of the input signal, at the low frequencies the general trend of the spectral behavior curve should be compared to the analytical system Bode, and not the exact numerical value.

The two dominant modes of the aircraft at the frequency range of interest, i.e., the phugoid at the low frequencies and the short period at the high frequencies are clearly characterized by the neural network as can be seen in all the spectral plots in Figures 8.2 and B.1, B.2.

Figure 8.3 and the corresponding Figures B.3 and B.4 in Appendix B. presents the same information as Figures 8.2, B.1 and B.2, but for the damaged aircraft trained in a separate neural network. The spectral results obtained for the undamaged aircraft hold for the damaged aircraft as well. In Figure 8.3 the damage effect is clearly seen in that there is no response of the aircraft to excitation due to the left stabilator (dstl) over the whole frequency range. The residual value of the network output at about -80dB in the lower part of Figure 8.3 is due to the initial weight randomization and numerical noise. The system Bode and the Desired value spectrum is minus infinity and thus do not appear on the graph.

Looking at all six figures, Figures 8.2, 8.3 and Figures B.1-B.4 some general observations from the results are emerging.

1. The network learns the transfer function of each of the six inputs. This can be seen by comparing the upper and lower graphs of each figure. Since each figure shows the corresponding right and left surfaces, the results should be symmetrical for both, apart from Figure 8.3 where the damage is seen. The slight differences between the right and left surfaces of each of the three pairs indicate the individual treatment of each input.

2. The network is able to resolve the similar input pairs for example left and right stabilators, leading or trailing edge flaps, although each is treated separately. This is seen by the very close resemblance of the spectrums of each pair, right and left inputs, which in the undamaged plant are identical, and also for the damaged plant except for the stabilators.

3. The network is clearly able to identify the damage in the second plant as seen in the lower part of Figure 8.3.

4. There is a very good agreement between the desired response and the network response over almost all the frequency range. The desired and network responses were calculated using the same random input sequence.

5. There is a very good agreement between the network response and the analytical Bode plot over about the upper two and a half decades. It can be said to have exact match over frequencies from 0.03 to 5 Hz. The differences are due to the random nature of the input signal used to generate the network response. The difference is also due to the difficulty to practically excite the very low frequencies, and the lower number of presentations of the phugoid mode relative to the short period mode.

6. The networks have learned the spectral behavior of the two plants as presented in Figures 8.2, 8.3, B.1-B.4.

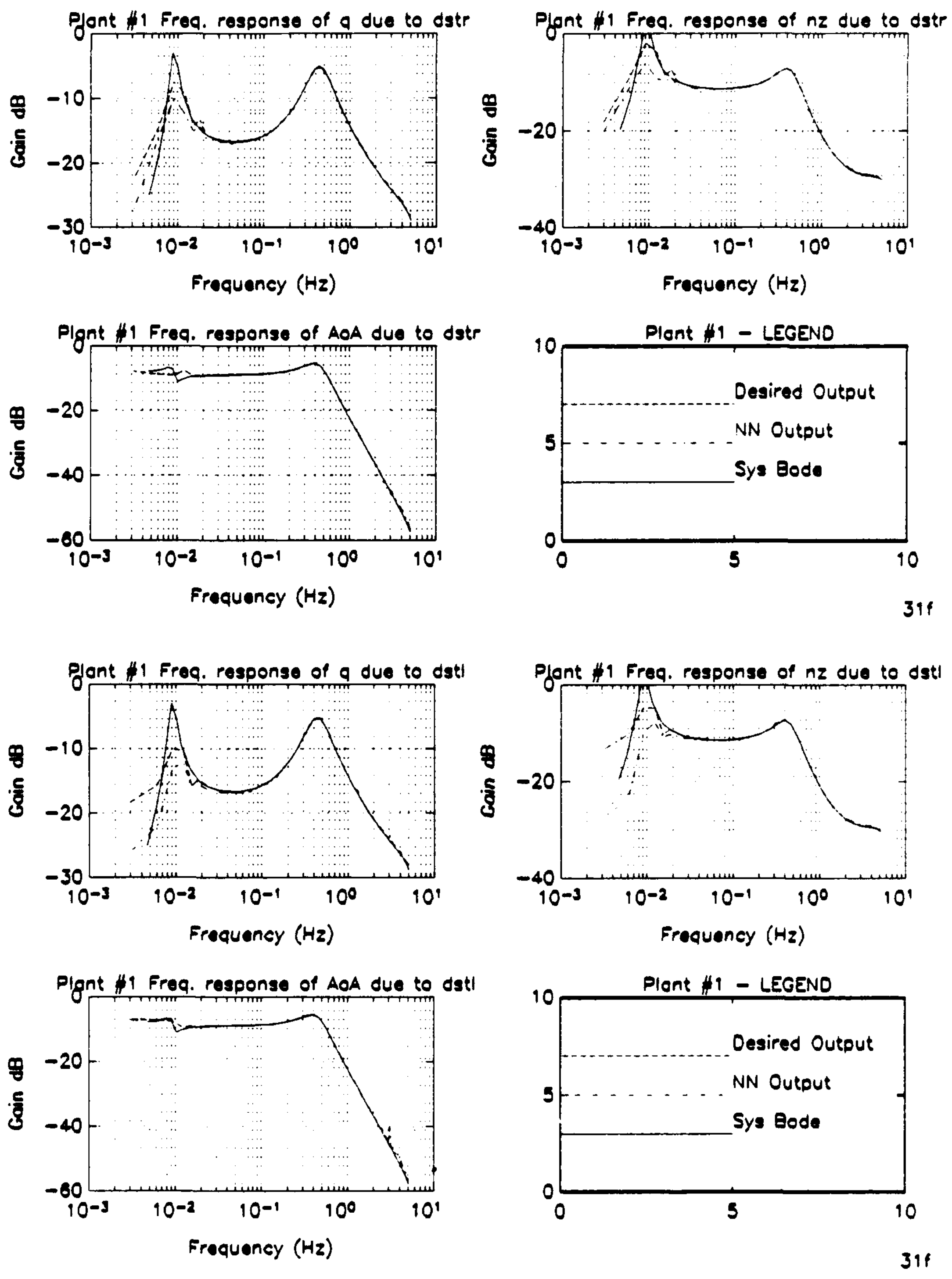
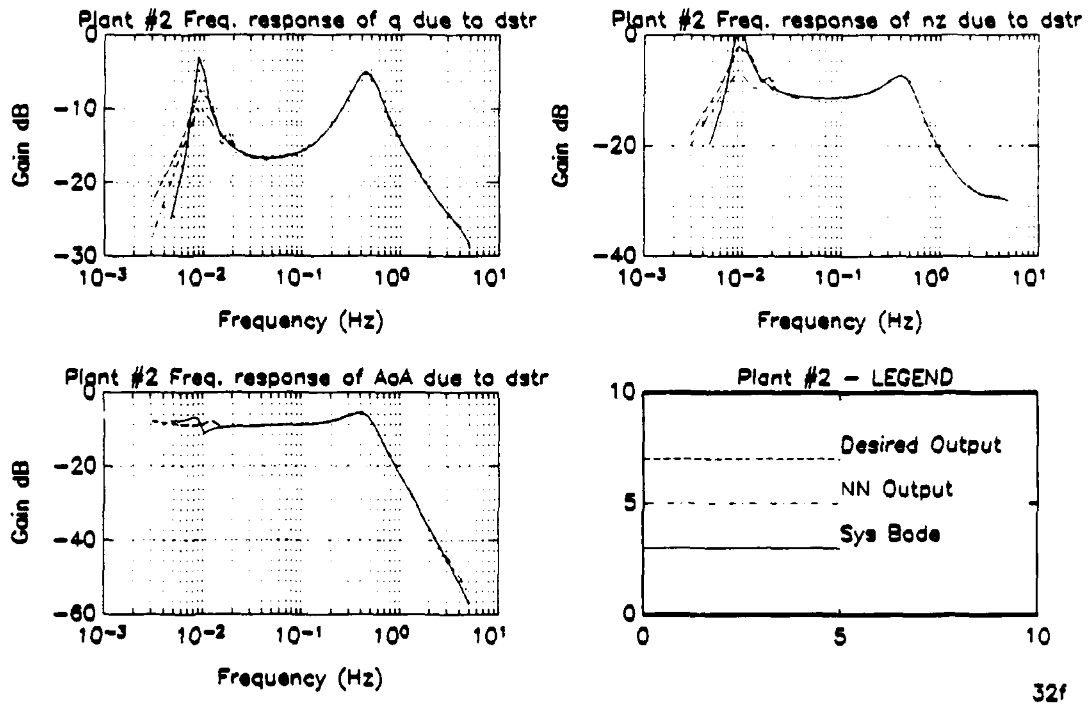
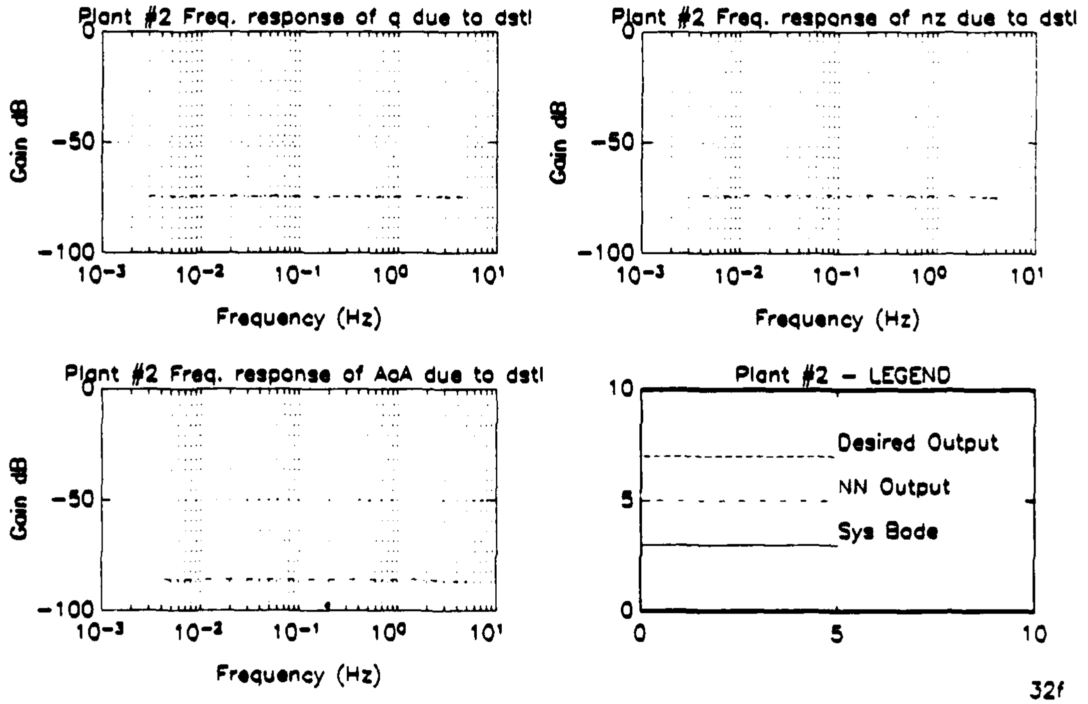


Figure 8.2: Spectral Response of Undamaged Aircraft Due to Stabilators



32f



32f

Figure 8.3: Spectral Response of Damaged Aircraft Due to Stabilators

2. Single Plant Emulation - Time Domain Results and Discussion

The time domain analysis shows the response of the outputs of the system, i.e., q, n_z *and* α, to a dipole test signal driving each input. Due to the multiplicity of inputs and outputs, and in order not to disrupt the flow of reading, representative figures follow here, and the rest are gathered in Appendix B. and referenced accordingly. The organization of all figures in the appendix is identical to the figures shown and described here.

Figure 8.4 shows the normal acceleration response, n_z, of the undamaged aircraft due to a dipole input command at the right stabilator dst_r (upper plots) and at the left stabilator dst_l (lower plots). The dipole presents a full deflection in the positive direction for one second followed by a full deflection command in the negative direction for a duration of another second, and then reset to zero deflection. The dotted line represents the dipole and is indicative of the switching times (the magnitude is ± 1 and out of scale). The solid line is the desired behavior, and the dashed line in the neural net output. The response of the normal acceleration due to leading edge flaps dle_r and dle_l, and due to trailing edge flaps dte_r and dte_l appear in Appendix B. Figures B.8-B.9 respectively. The pitch rate response of the undamaged aircraft to a dipole command in all inputs is in Figures B.5-B.8. The corresponding angle-of-attack response appears in Figures B.10-B.12.

Each graph shows the desired and network output for the command. The error between the two is presented in the lower plot of each pair. Note that the output are on the order of tenths while the errors are at the order of 10^{-3}. The relative error of all input-output relations is about 1%, which was the criterion for successful learning. It is interesting to see that the network which was trained by exciting all inputs simultaneously was equally trained on all inputs regardless of the individual behavior and amplitudes. Moreover, each input is interpreted individually as can be seen from the different error of similar control surfaces pairs (left and right), in Figures B.7, B.8 and B.9 (it is impossible to resolve this from the actual response because of the very small difference).

The damaged aircraft was trained on a separate network. Figure 8.5 presents the normal acceleration response of the damaged aircraft due to the stabilators. The graphs of n_z due to leading and trailing edge flaps appear in Figures B.16, B.17 respectively. the corresponding plots of the damaged aircraft pitch rate response to a dipole command in the various control surfaces are in Appendix B, Figures B.13 - B.15. The effects of the inputs on α - AoA, appear in Figures B.18 - B.20. One should be noted to the scale of the damaged stabilator output - 10^{-4} in Figure 8.5 which is about three orders of magnitude smaller than the right stabilator response. In the damaged stabilator it is not surprising that the error is at the same magnitude as the actual output since the desired value is zero. The output of 10^{-4} is effectively zero.

The capability of the network to react to individual input separately is even more emphasized in the damaged model. Although there is a totally different dynamics of the damaged input - left stabilator, as seen in Figure 8.5, it does not effect the training of the network to respond correctly to the rest of the inputs. Comparing Figures B.5-B.12 with B.13-B.20 one by one shows almost identical network behavior as was expected from the similarities between the two models. Thus the observations on the undamaged aircraft hold to the damaged aircraft as well.

Examining the results shown on Figures 8.4, 8.5 and Figures B.5-B.20 as a complete set describing the two networks used to emulate the undamaged and damaged aircraft separately, brings the following general observations.

1. The network resolves the relative influence of each input separately although while training in parallel the outputs included the cumulative contribution of all inputs.

2. The network is able to resolve the similarities in the left and right control surface pairs, although each is treated separately. This is seen by the very close resemblance of the responses of each pair of inputs, which in the undamaged plant are identical, and also for the damaged plant except for the stabilators.

3. The network is clearly able to identify the damage in the second plant as seen in Figures 8.5, B.13 and B.18 in comparison to Figures 8.4, B.5 and B.10.

4. There is almost perfect agreement between the desired response and the network response. Practically they can be considered identical. For most cases the difference is less than 1%.

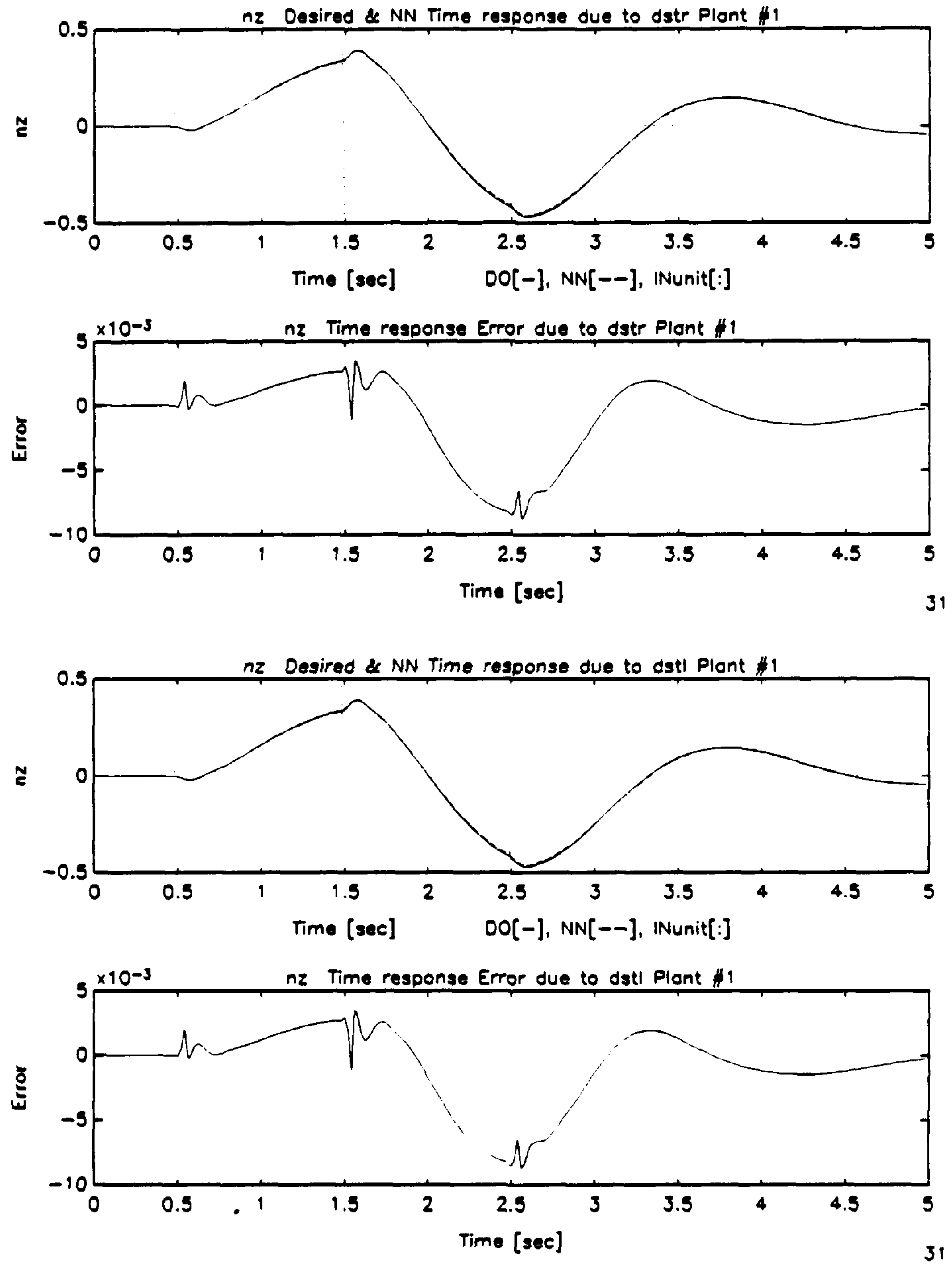
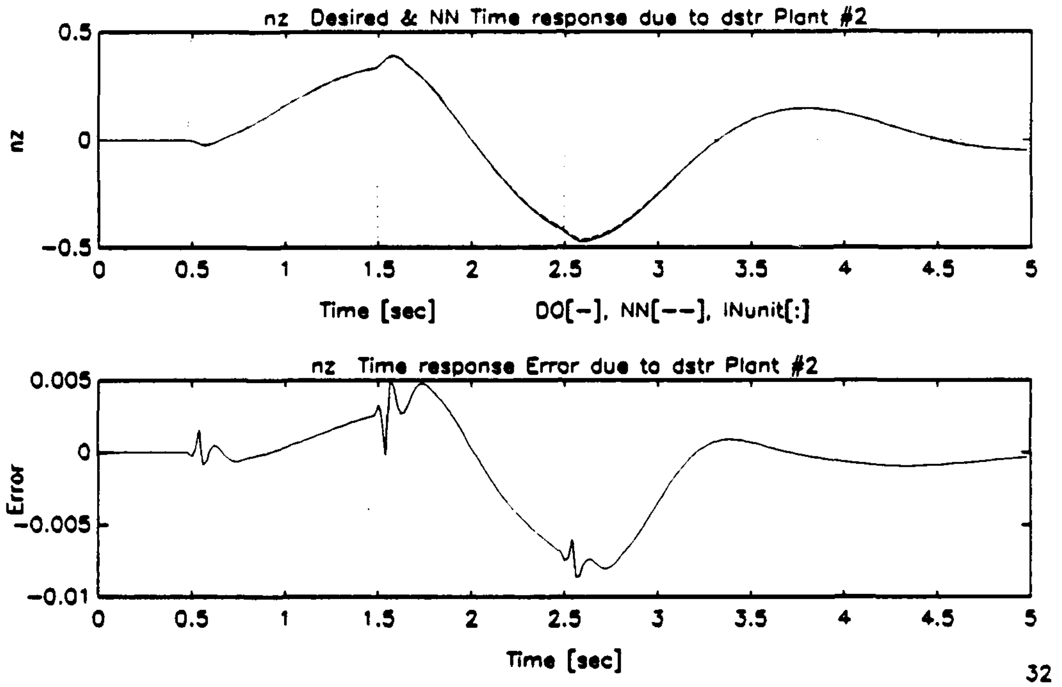


Figure 8.4: Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Command in the Stabilators



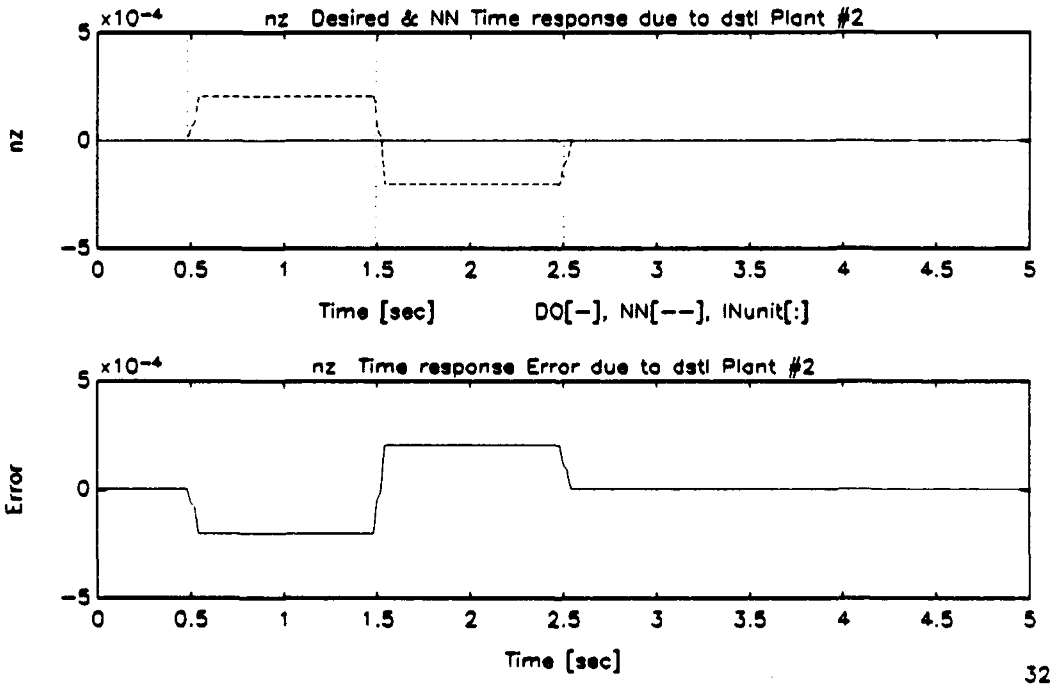


Figure 8.5: **Normal Acceleration Response of the Damaged Aircraft Due to Dipole Command in the Stabilators**

B. EMULATION OF MULTIPLE PLANTS IN A SINGLE NETWORK

For multiple-system emulation the same network structure as seen in Figure 8.1 is used for the results presented here, with $p1=1$, $p2=-1$ for undamaged plant and $p1=-1$, $p2=1$ for damaged plant. The network structure deliberately has had the same configuration as for a single plant emulation, so that comparisons can be made. Both the damaged and undamaged models were used to simultaneously train the single network via the procedure described in section IV.D. The results shown here are for the two 25th order models emulated together in a single network. It is important to mention that the procedure is general for any number of models, and two models were used here merely for demonstration, without any restrictions.

1. Multiple-Plant Emulation - Frequency Domain Results and discussion

The examination starts with spectral analysis of the network compared to the desired model. Figure 8.6 shows the spectral response of the pitch rate q, the normal acceleration n_z and the angle of attack AoA of the undamaged aircraft due to the right stabilator dst_r (upper plots) and the left stabilator dst_l (lower plots). Each graph shows the analytic frequency response in the solid line, which is the Bode plot of the outputs due to the corresponding input, together with the desired outputs in the dashed line, and network output values in the dash-dot line. Analogous plots for the leading edge and trailing edge control surfaces of the undamaged plant, appear in Appendix B, Figures B.21 and B.22 correspondingly. Figure 8.7 and the corresponding Figures B.23 and B.24 in Appendix B. presents the same information as Figures 8.6, B.21 and B.22, but for the damaged aircraft *trained on the same network* used for the undamaged aircraft.

The results from the spectral analysis obtained for the single-plant emulation for both undamaged and damaged aircraft hold for multiple plants emulated in a single network as well. The main results from Figures 8.6, 8.7, B.21 and B.22 show almost perfect match between the system Bode plot, the desired output and network output frequency response over the range of 0.03 Hz to 5 Hz. In Figure 8.7 the damage is clearly seen where there is no response of the aircraft to excitation due to the left stabilator (dstl) over the whole frequency range.

Looking at Figures 8.6, 8.7 and Figures B.21-B.24 some general observations from the results are emerging.

1. The network learns the transfer function of each input individually.

2. The unified network learns the full input-output transfer function of each plant individually.

3. The network is able to resolve the similar input pairs such as left and right stabilators, leading and trailing edge flaps, although each is treated separately. This is seen by the very close resemblance of the spectrums of each pair, right and left inputs, which in the undamaged plant are identical, and also for the damaged plant except for the stabilators. This holds also for comparing the corresponding undamaged and damaged plots, i.e., the right stabilator part of Figures 8.6 and 8.7, as well as comparing Figure B.21 to B.23 and B.22 to B.24.

4. The results of the multi-plant network are very similar compared to the results obtained from the single-plant network. Differences are due to the same reasoning given for the single-plant network in VIII.A.1.a.

5. The network is clearly able to identify the damage in the second plant as seen in Figure 8.7.

6. There is a very good agreement between the network response and the analytical Bode plot over frequencies from 0.03 to 5 Hz. The differences are due to the insufficient persistent excitation in the input signal.

7. The single network has learned the spectral behavior of the two plants as presented in Figures 8.6, 8.7, B.21-B.24.

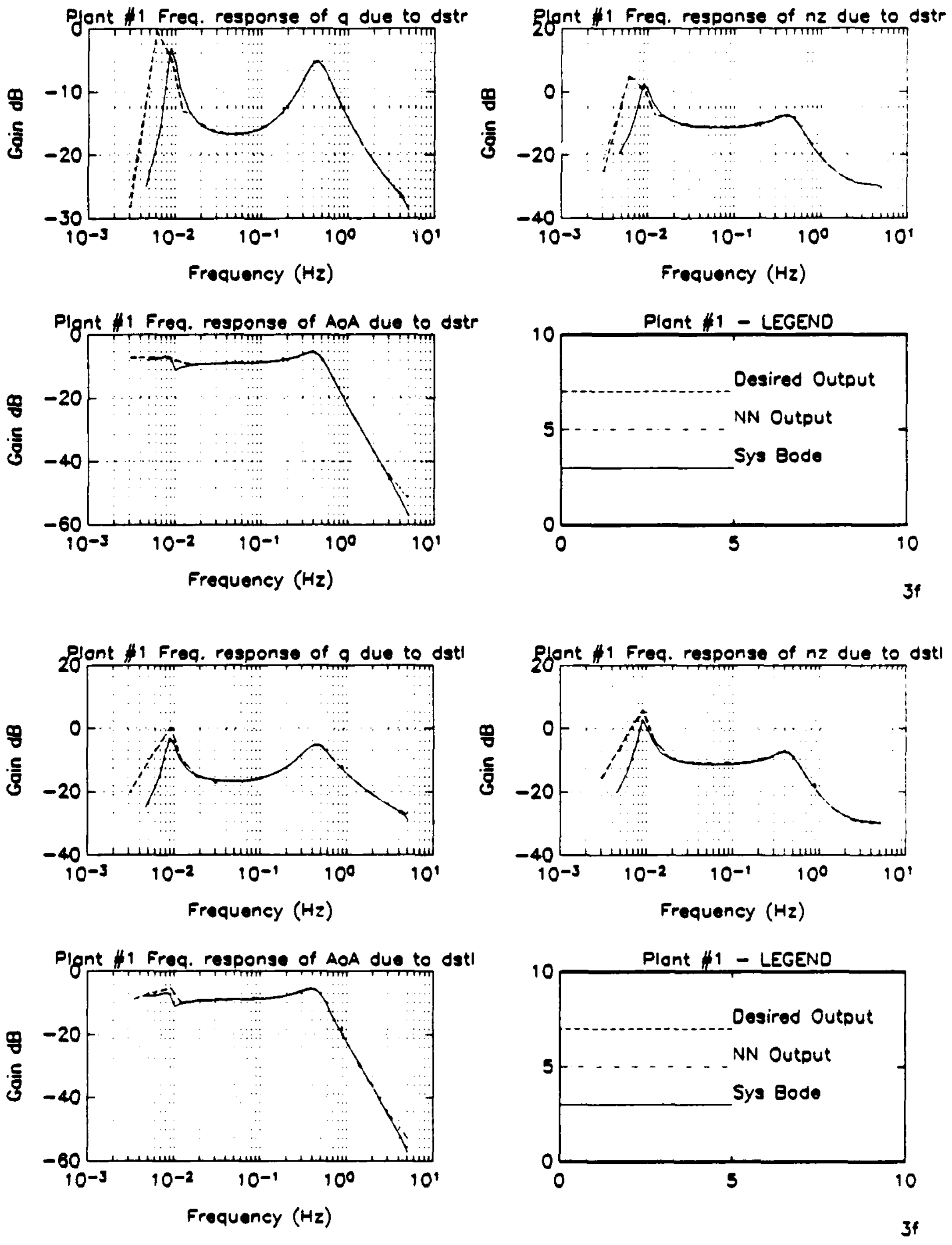


Figure 8.6: **Spectral Response of Undamaged Aircraft Due to Stabilators in a Multiple-Plant Single Network Emulation**

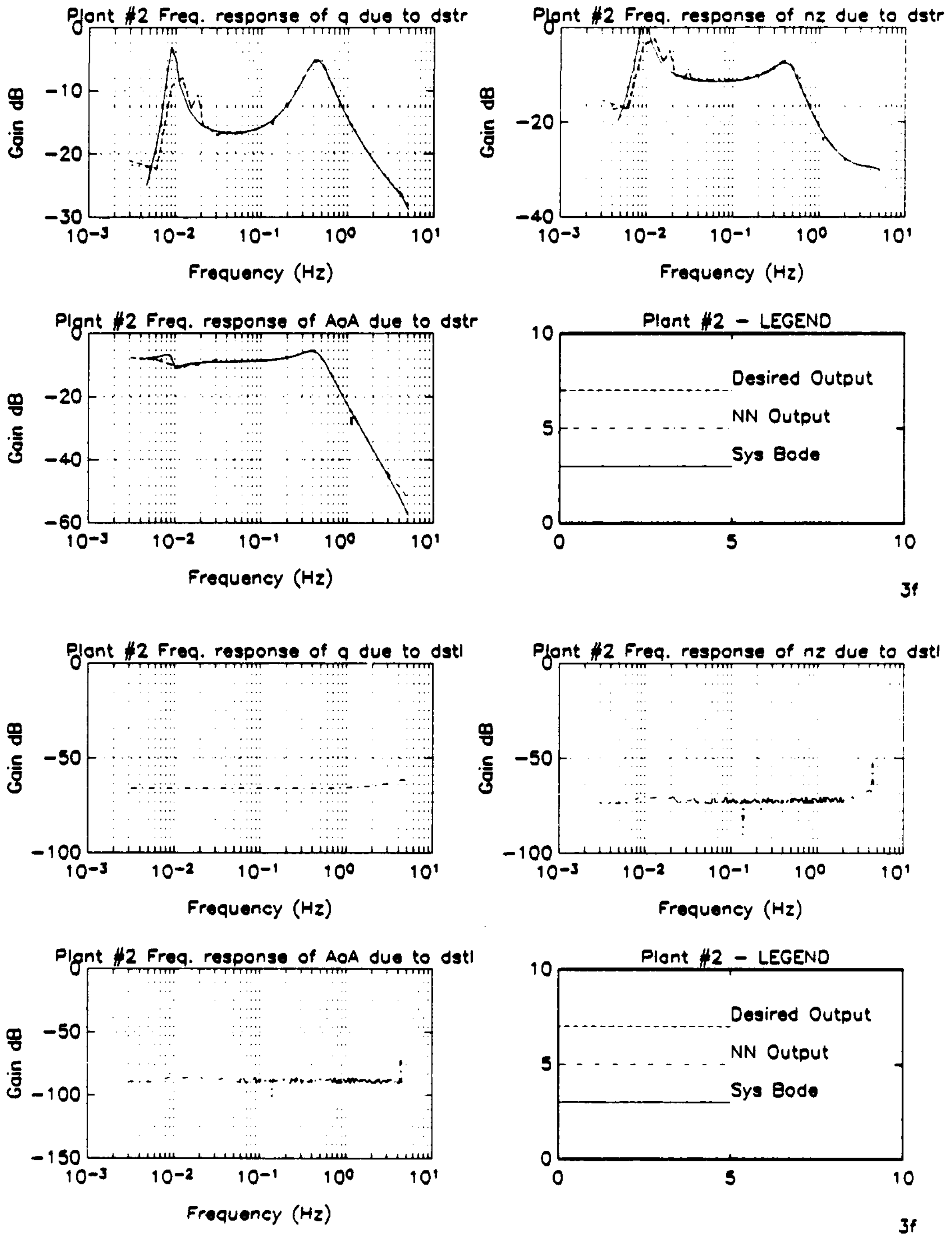


Figure 8.7: Spectral Response of Damaged Aircraft Due to Stabilators in a Multiple-Plant Single Network Emulation

2. Multiple-Plant Emulation - Time Domain Results and Discussion

The time domain analysis shows the response of the outputs of the system, i.e., q, n_z *and* α, to a dipole test signal driving each input. Due to the multiplicity of inputs and outputs, and in order not to disrupt the flow of reading, representative figures follow here, and the rest are gathered in Appendix B, and referred accordingly. The organization of all figures is identical to the ones shown and described here.

Figure 8.8 shows the normal acceleration response, n_z, of the undamaged aircraft due to a dipole input command at the right stabilator dst_r (upper plots) and at the left stabilator dst_l (lower plots). The dotted line represents the dipole and is indicative of the switching times (the magnitude is ± 1 and is out of scale). The solid line is the desired behavior, and the dashed line in the neural net output. The response of the *normal acceleration due to* leading edge flaps dle_r and dle_l, and due to trailing edge flaps dte_r and dte_l appear in Appendix B, Figures B.28-B.29 respectively. The pitch rate response of the undamaged aircraft to a dipole command in all inputs is in Figures B.25-B.28. The corresponding angle-of-attack response appears in Figures B.30-B.32.

The damaged aircraft was trained simultaneously on the same network. Figure 8.9 presents the normal acceleration response of the damaged aircraft due to the stabilators. The graphs of n_z due to leading and trailing edge flaps appear in Figures B.36, B.37 respectively. The corresponding plots of the damaged aircraft pitch rate response to a dipole command in the various control surfaces are in Appendix B, Figures B.33 - B.35. The effects of the inputs on α - AoA, appear in Figures B.38 - B.40. One should be noted to the scale of the damaged stabilator output - 10^{-3} in Figure 8.9 which is about three orders of magnitude smaller than the right stabilator response.

Each graph shows the desired and network output for the command. The error between the two is presented in the lower plot of each pair. Note that the output errors are at the order of 10^{-2}. The relative error of input-output relations is from about 5% down to 2%. It is interesting to see that the network which was trained by exciting all inputs simultaneously was equally trained on all inputs and both plants, regardless of the individual behavior and amplitudes.

Examining the results shown on Figures 8.8, 8.9 and Figures B.25-B.40 as a complete set describing the single network used to emulate the undamaged and damaged aircraft together, brings the following general observations.

1. The results obtained for the single-plant network generally apply to the multiple-plant emulation as well as shown in Figures 8.8, 8.9, and Figure B.25-B.40.

2. The output errors are greater than the errors achieved using a single plant emulation, and valued at about 2-5%. The differences are due to the greater complexity of the overall transfer function the network represents.

3. The network is clearly able to distinguish the damaged plant from the undamaged model as of Figures 8.9, B.33 and B.38.

4. There is almost perfect agreement between the desired response and the network response. Practically they can be considered identical.

5. A single network is capable of emulating multiple systems with only a very small loss of accuracy due to the greater complexity.

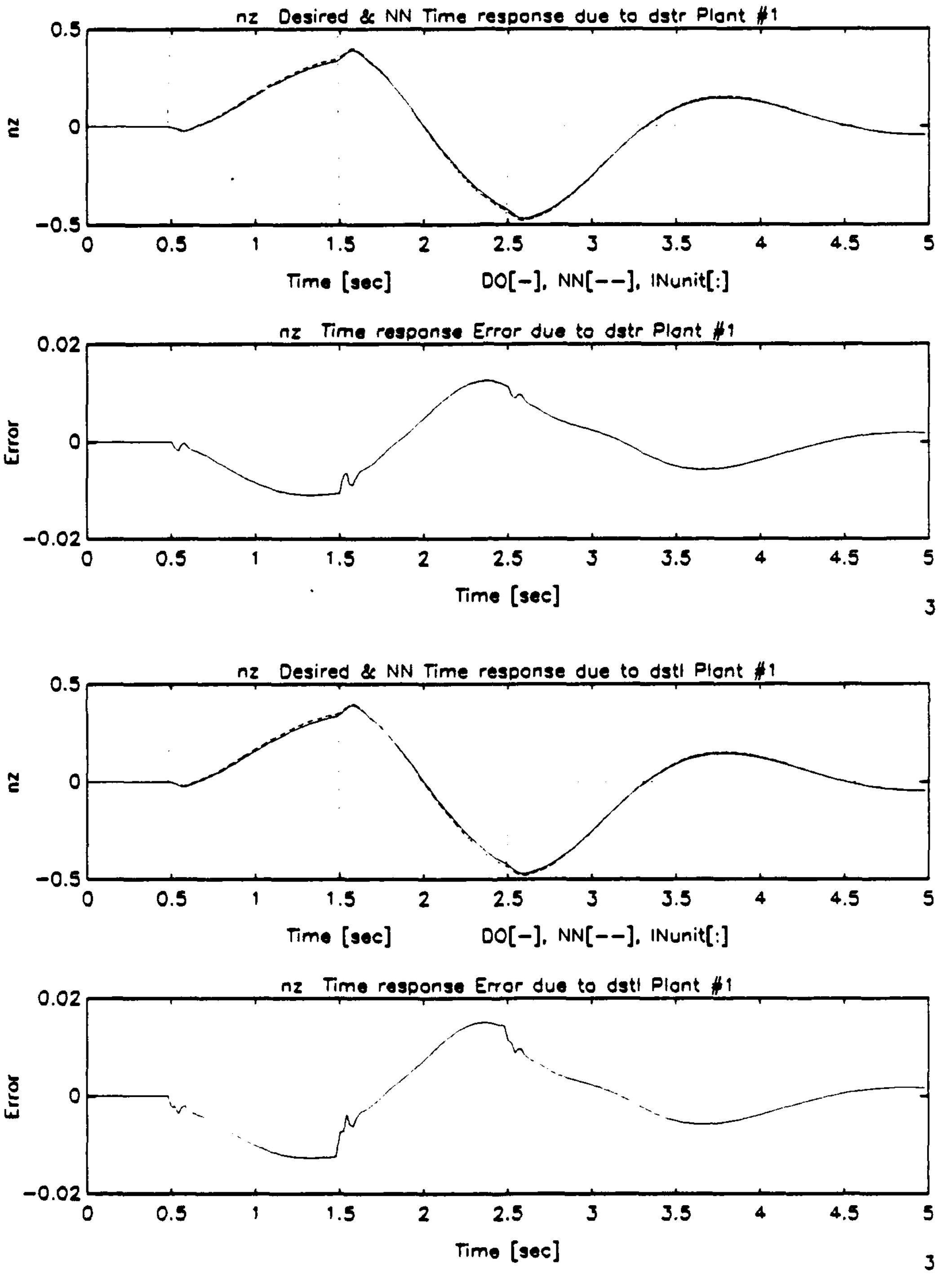


Figure 8.8: **Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Command in the Stabilators in a Multiple-Plant Network**

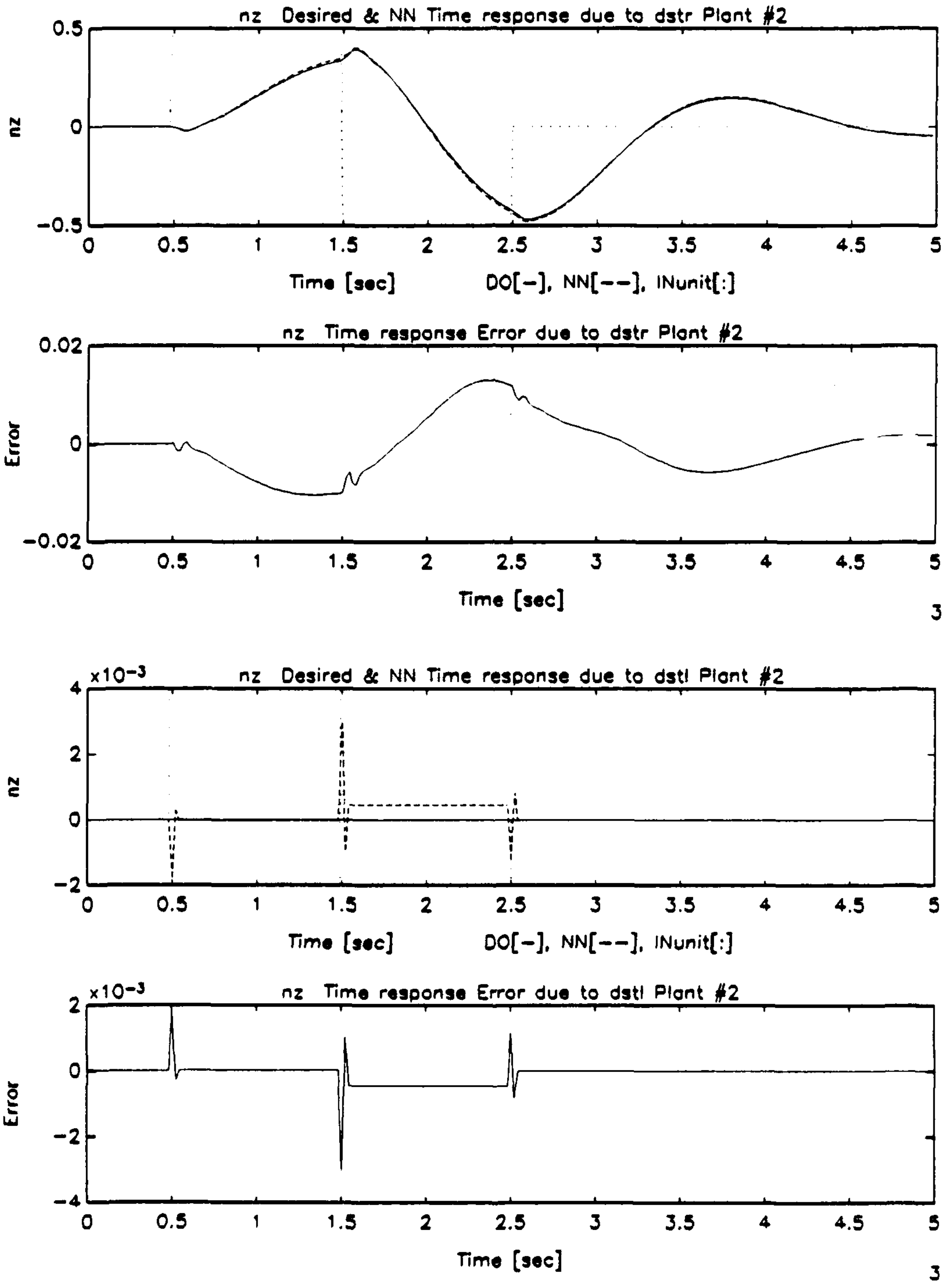


Figure 8.9: **Normal Acceleration Response of the Damaged Aircraft Due to Dipole Command in the Stabilators in a Multiple-Plant Network**

Figure 8.10 shows the normal acceleration response, n_z, due to a random binary excitation of all inputs simultaneously. At the point indicated on the middle of the graph there is a transition from one model to the other. Observe that the time response is continuous in both cases. Each graph shows the desired and network output for the command five seconds before and after the switch. The error between the two is presented in the lower plot of each pair. Again plant #1 is the undamaged aircraft and plant #2 is the impaired model. The corresponding graphs for the pitch rate q and angle of attack α appear in Appendix B, Figures B.41 - B.42.

The observations from the results in Figures 8.10, B.41 and B.42 are the following.

1. There is a smooth continuous behavior of the output variables at the transition.
2. The output error for a particular output is at the same level for both plants.
3. The network is clearly able to distinguish the damaged plant from the undamaged model and respond accordingly.
4. There is almost perfect agreement between the desired response and the network response when exciting all six control surfaces. This means that the network has learned the overall complex transfer function from the six inputs to the three outputs altogether and not only per each individual input.
5. The error at the transition point is slightly higher due to the regression, until the layer fills with data corresponding to the current model. Nevertheless, the error at the switch is at the same order of magnitude as in the normal operation.

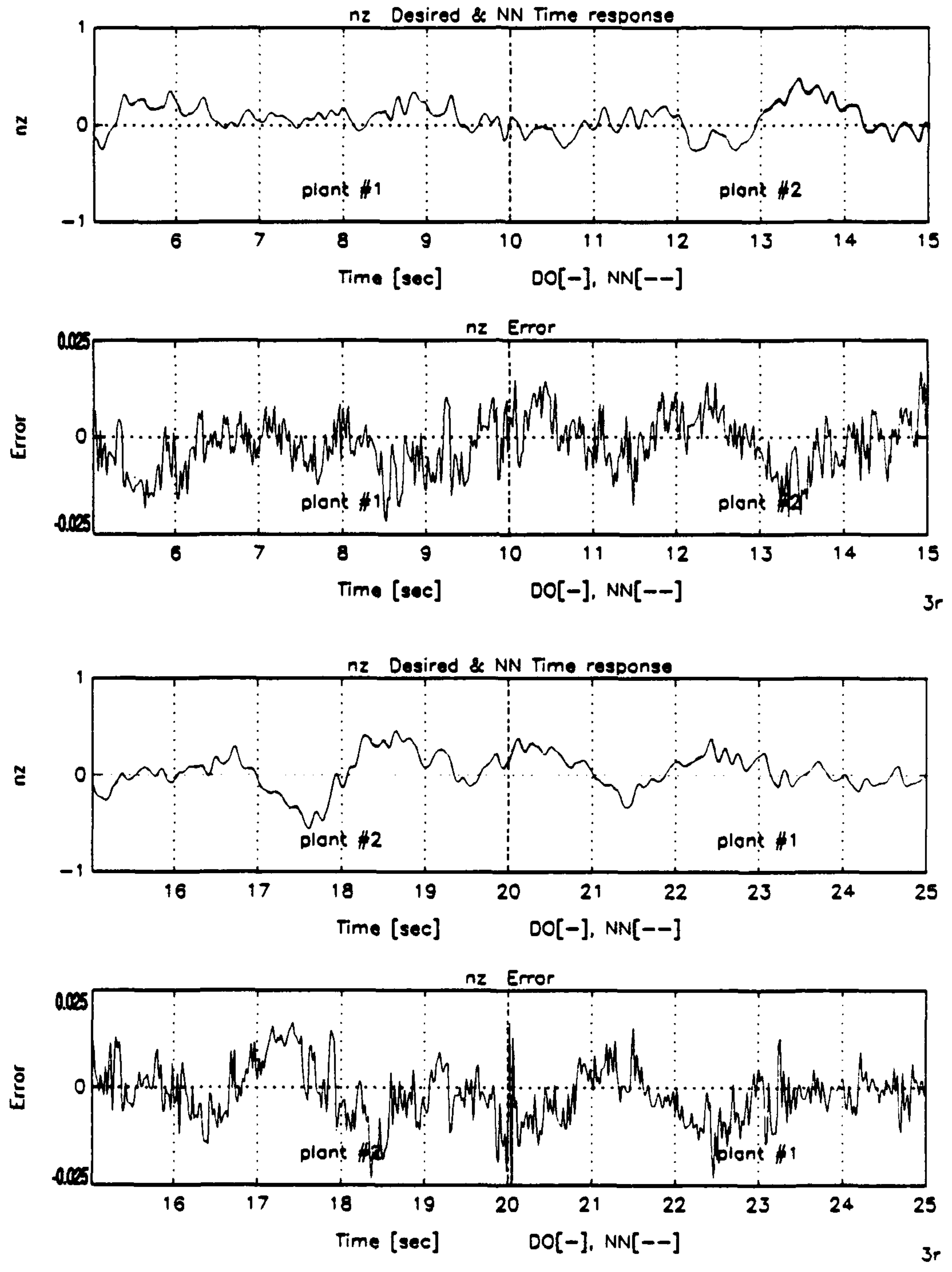


Figure 8.10: **Normal Acceleration Response of the Single Network Emulating Two Plants Due to RBS Excitation of All Inputs**

C. MINIMAL REALIZATION OF A SINGLE-NETWORK MULTIPLE-PLANT EMULATOR

It has been shown that a single ANN is capable of emulating multiple-plants. A minimal realization of the multiple-plant network is now derived. The initial network shown in Figure 8.1 has two sub-networks, the forward on the left and the recurrent on the right. Each sub-network has two hidden layers with each hidden layer constructed of 36 PEs. This structure was successfully trained to emulate the two different aircraft models. This network served as the starting point for the minimal realization procedure.

The regression layers, *ComReg* and *OutReg* of Figure 8.1, were already reduced based on observation of the decreasing valued of the regressed weights of more prior samples. The singular values of the weight matrices from the regression layers and through successive hidden layers were analyzed. Figure 8.11 shows the singular value plot of the initial weight matrix (36×36), connecting the two hidden layers of the recurrent sub-network. There are three dominant singular values and the rest are spread below 0.1. Close examination of Figure 8.11 shows that there are more significant singular values hiding in the initial range due to the random initialization. The number of PEs needed was estimated using equations (6.2), (6.3) with variation ratio of $\nu=500$. The resultant layer needs only nine PEs. The singular values of the weight matrix connecting the same hidden layers in the minimal network are traced, as the network trains, in Figure 8.12. It can be seen that the same three dominant singular values appear in the minimal network, with the other six spread over the initial values. A lower variation ratio would have predicted less PEs. It is the choice of the designer to set the threshold, with post-training examination of the achieved accuracy.

A similar procedure was followed for the other three hidden layers. The singular values plots of the other weight matrices appear in Appendix C. On the recurrent side, the singular values of the weight between *OutReg* and *outh1* appear in Figures C.5 before, and in Figure C.6 after training with similar behavior.

In the forward sub-network, the singular values of the weight matrix between *ComReg* and *comh1* before minimization appear in Figure C.1, with values spread up to 2.5. Using the same variation ratio, $\nu=500$, in equation (6.3) yields an estimation of 20 PEs out of 36. The singular values of the minimized network are shown in Figure C.2, showing that the selection was quite cautious, and a smaller number might have been good enough for successful training. The Singular values of the weight matrix between *comh1* and *comh2* are traced in Figures C.3 and C.4 respectively.

The minimized network appear in Figure 8.13. The squares represent the processing elements. According to the current input value and the weights connecting to a PE the output value is viewed by size and color, converted here to shades of grey. It is seen that the activation from one layer to another is transferred differently by the PEs. This is clearly seen in *ComReg* and *comh1* layers. Given an input in terms of control commands at time $(t=k)$ at the *In* layer of Figure 8.13, and a model selection at the input nodes denoted *p1* or *p2*, the network is trained to generate an output of the selected aircraft model at time $(t=k+1)$.

Looking at the singular values plots in Figures 8.11, 8.12 and C.1-C.6, the following observations are drawn.

1. The singular values plot express the number of PEs carrying the activation levels between two successive layers.

2. The singular values plots trace the dynamics of the training, and may supply an estimation on when the network is trained, by the fact that the singular values converged to their final values. The slopes of the curves need to be divided by the temporal learning rate to show convergence. Figure 8.11 reflect convergence after about 500k iterations, whereas in the minimized network, as seen in Figure 8.12, after about 800k the network is not well converged yet.

3. The plots should be carefully examined for small but important singular values that might hide in the initial randomization values. This has happened i.e., in Figure C.3 and extracted also using equations (6.2) and (6.3).

4. The network should be trained long enough for the number of significant singular values to become evident, depending of course on the learning rate and initial randomization. This phenomenon is clearly seen in Figure 8.11 and 8.12 where the third dominant singular value emerge after about 100k iterations.

5. The singular values plots act as a feature detector, and can give physical explanation to the training characteristics of the network. In the current example the third dominant singular value in Figures 8.11, 8.12 C.5 and C.6, probably can be associated with the phugoid mode, which needed more iterations to have a significant number of presentations (about 10) for the network to respond to that mode. Once the network sensed this mode the adaptation of the weight to include the phugoid mode information was rather quick. The short period mode was probably related to the quick rising singular values.

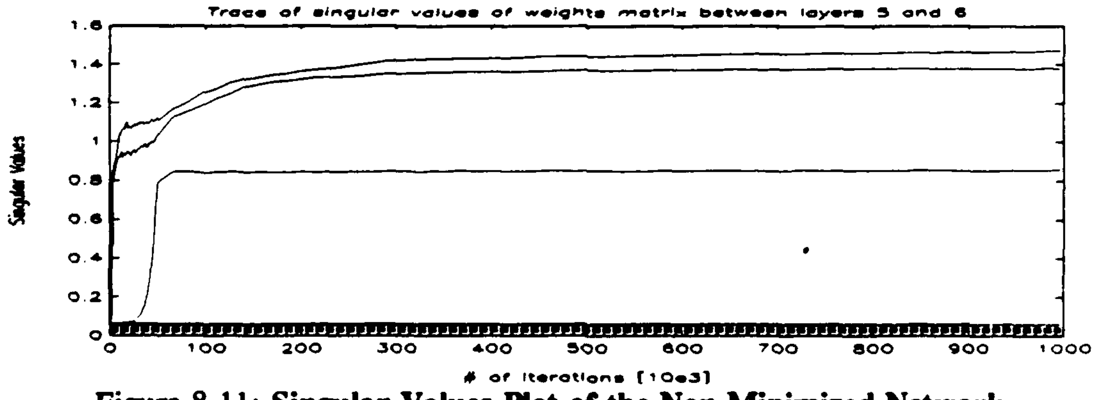


Figure 8.11: Singular Values Plot of the Non-Minimized Network

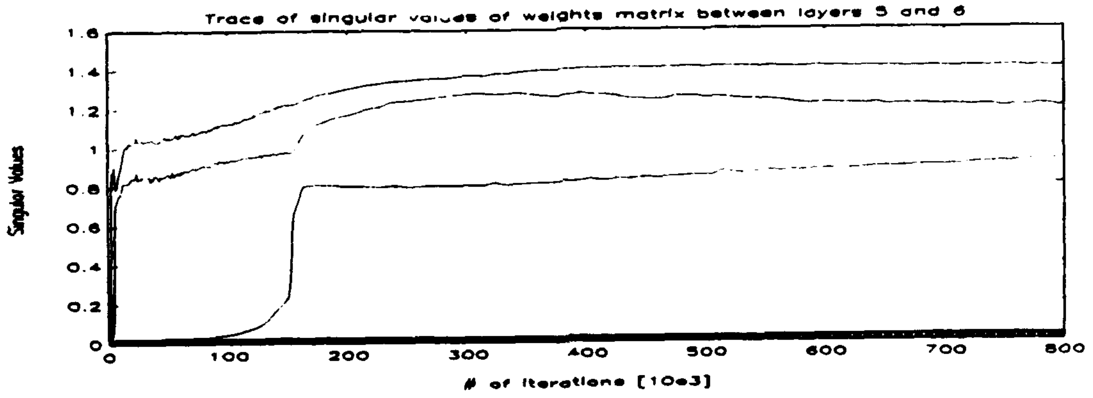


Figure 8.12: Singular Values Plot of the Minimized Network

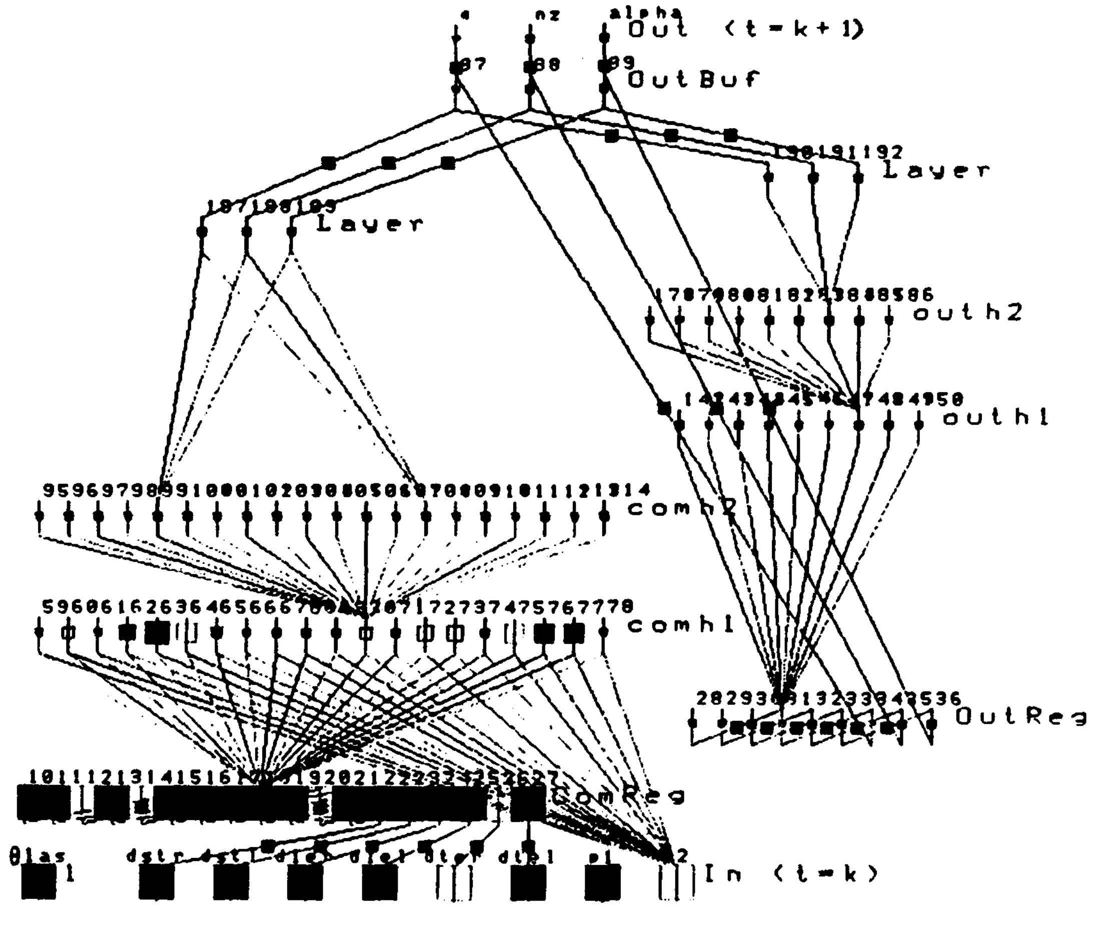


Figure 8.13: Minimal Realization of a Network for Multiple-Plant F/A-18 Emulation

1. Minimal Realization of a Multiple-Plant Network - Frequency Domain Results and Discussion

The spectral analysis compares the network with the desired model and with the performance of the non-minimized network. Figure 8.14 shows the spectral response of the pitch rate q, the normal acceleration n_z and the angle of attack *AoA* of the undamaged aircraft due to the right stabilator dst_r (upper plots) and the left stabilator dst_l (lower plots). Each graph shows the analytic frequency response in the solid line, which is the Bode plot of the outputs due to the corresponding input, together with the desired outputs in the dashed line, and network output values in the dash-dot line. Analogous plots for the leading edge and trailing edge control surfaces effects on the undamaged plant, appear in Appendix B, Figures B.43 and B.44 correspondingly. Figure 8.15 and the corresponding Figures B.45 and B.46 in Appendix B, presents the same information as Figures 8.14, B.43 and B.44, but for the damaged aircraft *trained on the same minimized network* used for the undamaged aircraft.

The results from the spectral analysis obtained for the multiple-plant emulation of both undamaged and damaged aircraft in a single network, summarized in section B, hold for the minimized network emulator as well. The main results from Figures 8.14, 8.15, B.43 and B.44 show almost perfect match between the system Bode plot, the desired output and network output frequency response over the range of 0.03 Hz to 5 Hz. In Figure 8.15 the damage is clearly seen where there is no response of the aircraft to excitation due to the left stabilator (dstl) over the whole frequency range. The fluctuations seen in Figure 8.15 are due to initial randomization, numerical noise and the random nature of the test signal. The values are at the order of -80dB, which can be neglected.

The results here are to be compared to the multiple-plant emulation in Figures 8.6,8.7 and Figures B.21 - B.24 correspondingly. The combined network is representing now a minimal network realization of the non-linear functional relationship between the inputs and outputs of the network that describes the damaged and undamaged aircraft.

Looking at Figures 8.14, 8.15 and Figures B.43-B.44, some general observations from the results are drawn.

1. The observations for the Multiple-plant emulation generally apply to the minimal realized multiple-plant emulation as well. There are instances that the minimized network shows even better results over the spectrum, like in the leading edge surfaces effect on both undamaged and damaged aircraft, as seen in comparing Figures B.21 for the multiple-plant emulation, and in Figure B.43 for the minimal realization. The improved behavior is probable due to the elimination of excess PEs and excessive connections in the minimized neural network.

2. The single minimized network has learned the spectral behavior of the two plants.

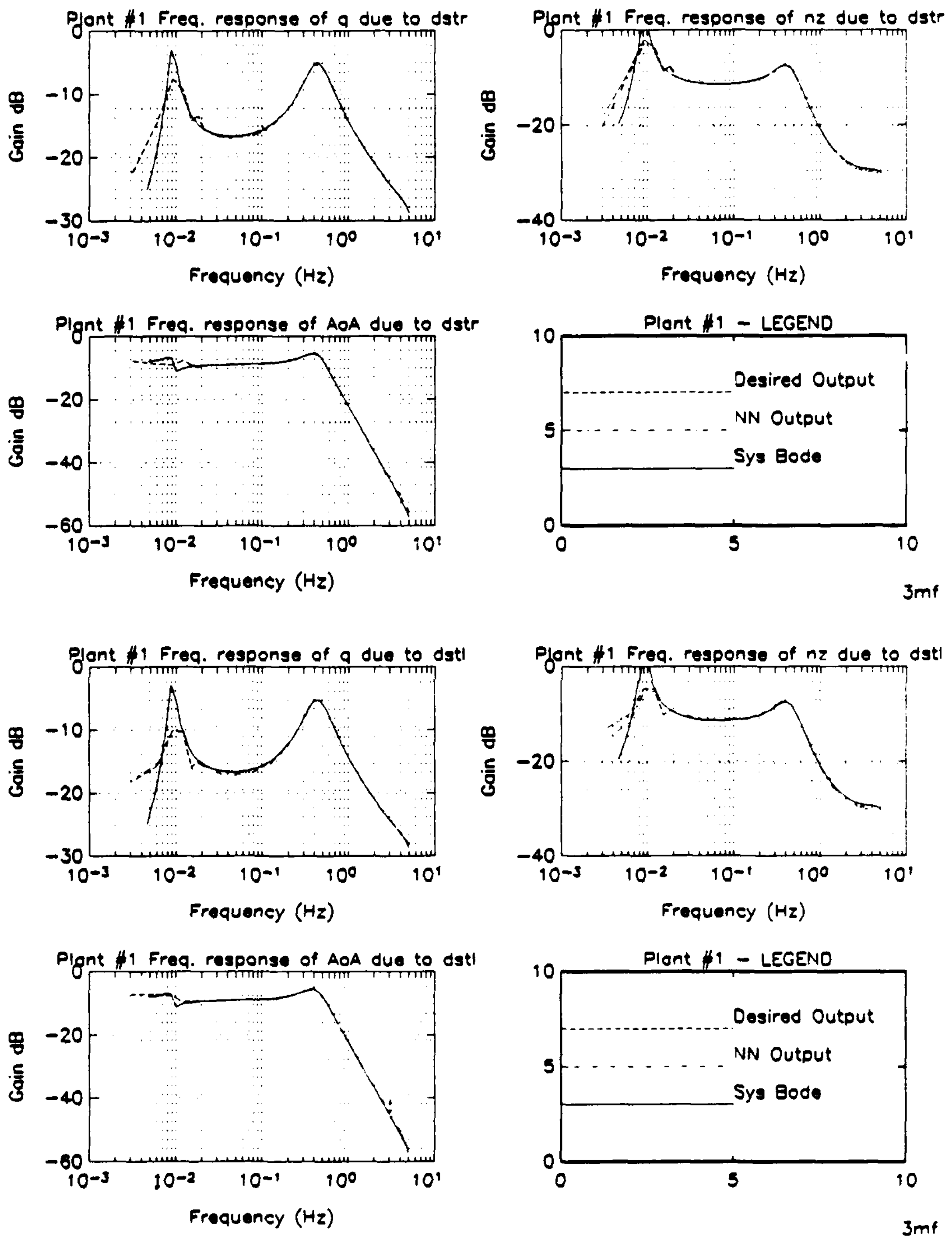


Figure 8.14: Spectral Response of Undamaged Aircraft Due to Stabilators in a Minimally Realized Multiple-Plant Single Network Emulator

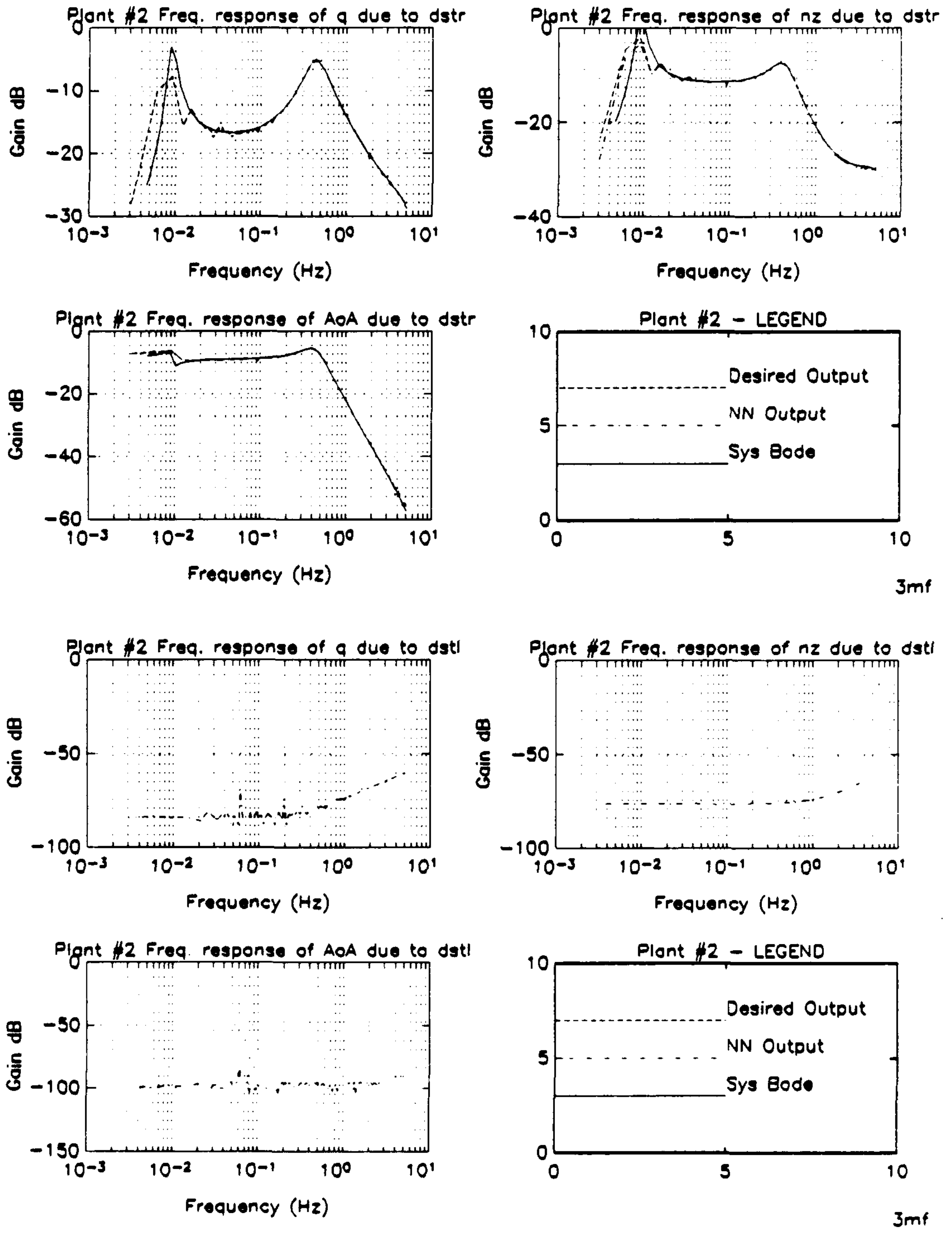


Figure 8.15: **Spectral Response of Damaged Aircraft Due to Stabilators in a Minimally Realized Multiple-Plant Single Network Emulator**

2. Minimal Realization of a Multiple-Plant Network - Time Domain Results and Discussion

The time domain analysis compares the response of the outputs of the system, i.e., q, n_z *and* α, with the network reaction to a dipole test signal driving each input. Figure 8.16 shows the normal acceleration response, n_z, of the undamaged aircraft due to a dipole input command at the right stabilator dst_r (upper plots) and at the left stabilator dst_l (lower plots). The dotted line represents the dipole and is indicative of the switching times (the magnitude is ± 1 and is out of scale). The solid line is the desired behavior, and the dashed line in the neural net output. The response of the normal acceleration due to leading edge flaps dle_r and dle_l, and due to trailing edge flaps dte_r and dte_l appear in Appendix B. Figures B.50-B.51 respectively. The pitch rate response of the undamaged aircraft to a dipole command in all inputs is in Figures B.47-B.49. The corresponding angle-of-attack response appears in Figures B.52-B.54.

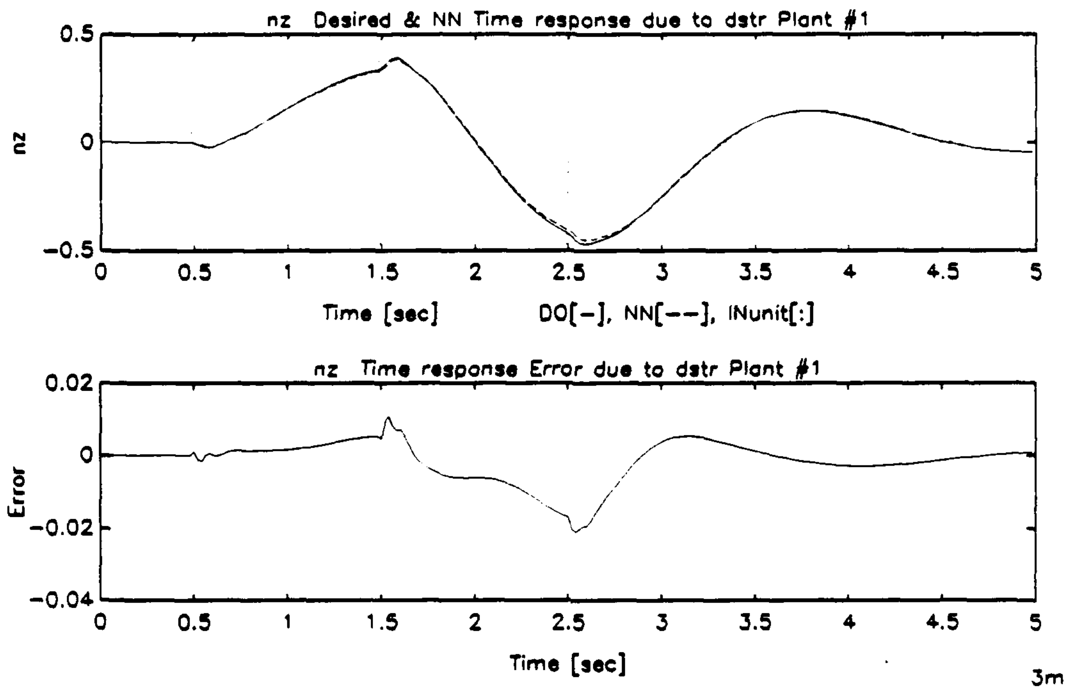
The damaged aircraft was trained simultaneously on the same minimized network. Figure 8.17 presents the normal acceleration response of the damaged aircraft due to the stabilators. The graphs of n_z due to leading and trailing edge flaps appear in Figures B.58, B.59 respectively. The corresponding plots of the minimized network representation of the damaged aircraft pitch rate response to a dipole command in the various control surfaces are in Appendix B, Figures B.55 - B.57. The effects of the inputs on α - AoA, appear in Figures B.60 - B.62. One should be noted to the scale of the damaged stabilator output - 10^{-3} in Figure 8.17 which is about three orders of magnitude smaller than the right stabilator response.

Each graph shows the desired and network output for the command. The error between the two is presented in the lower plot of each pair. Note that the output errors are at the order of 10^{-2}. The relative error of input-output relations is about 5% at most down to 1%. The network was trained by exciting all inputs simultaneously and equally performs on all inputs of both plants, regardless of the individual behavior and amplitudes.

Examining the results shown on Figures 8.16, 8.17 and Figures B.47-B.62 as a complete set describing the single minimal network used to emulate the undamaged and damaged aircraft together, and comparing to the non-minimal single network for multiple-plant emulation in Figures 8.8,8.9 and Figures B.25-B.40 correspondingly, brings the following general observations.

1. The results obtained for the single network of multiple-plant generally apply to the minimally realized single network of multiple-plant emulation as well.

2. The output errors are at the same level as the errors achieved using a mom-minimal single network for multiple-plant emulation, and valued at about 1-5%.



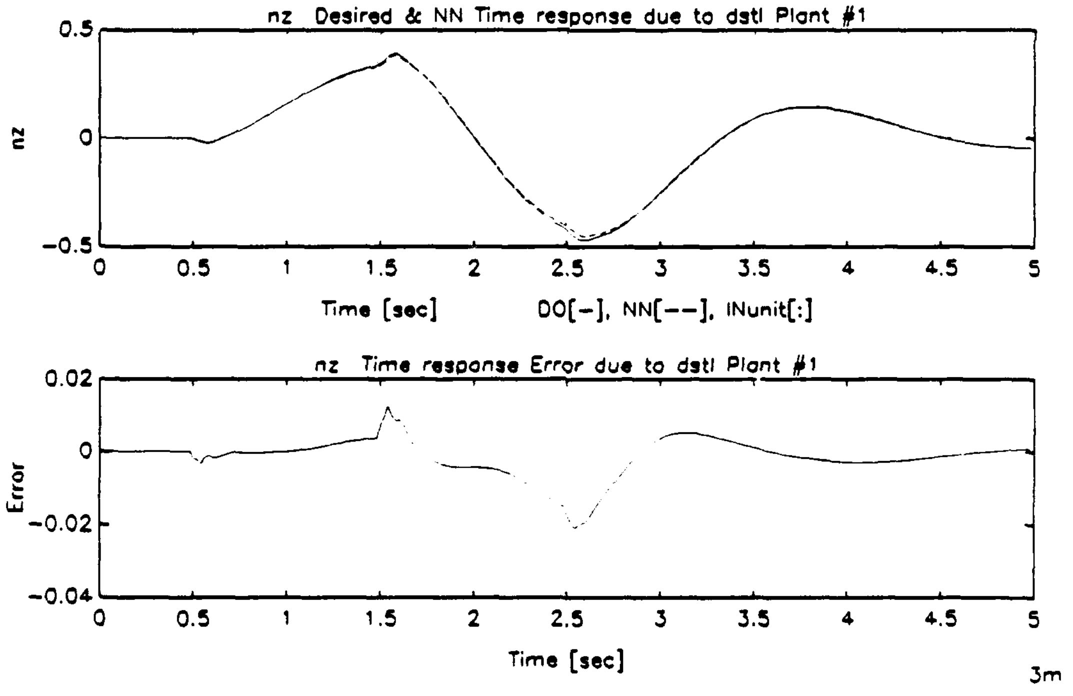


Figure 8.16: **Normal Acceleration Response of the Undamaged Aircraft Due to Dipole Stabilators Command in a Minimized Multiple-Plant Network**

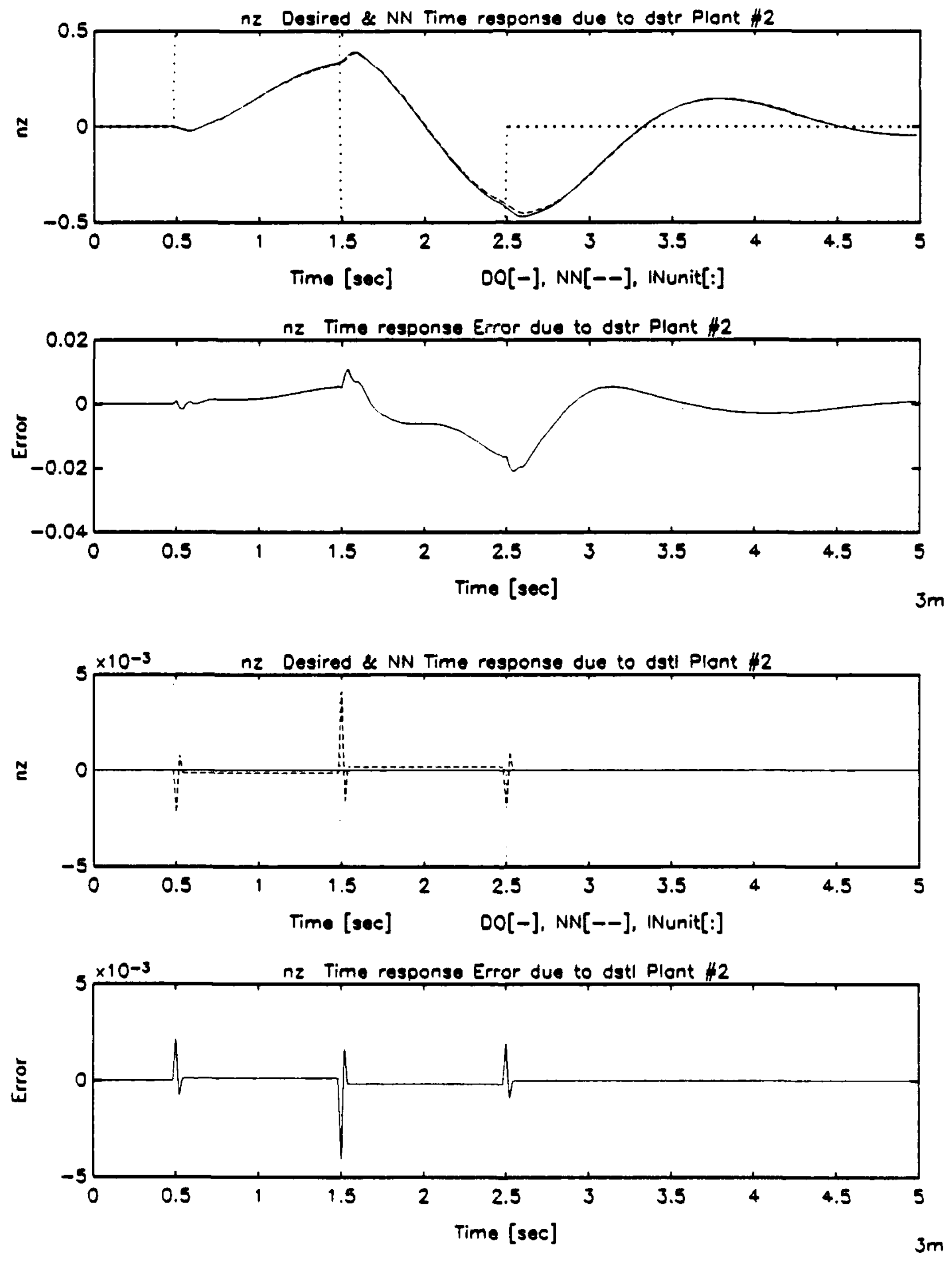


Figure 8.17: Normal Acceleration Response of the Damaged Aircraft Due to Dipole Stabilators Command in a Minimized Multiple-Plant Network

Figure 8.18 shows the normal acceleration response, n_z, due to a random binary excitation of all inputs simultaneously. At the point indicated on the middle of the graph there is a transition from one model to the other. Each graph shows the desired and network output for the command five seconds before and after the switch. The error between the two is presented in the lower plot of each pair. Plant #1 is the undamaged aircraft and plant #2 is the impaired model. The corresponding graphs for the pitch rate q and angle of attack α appear in Appendix B, Figures B.63 - B.64. The results here are to be compared to the non-minimal single network for multiple-plant emulation in Figure 8.10 and Figures B.41 - B.42 correspondingly.

The observations from the results in Figures 8.18, B.63 and B.64 in comparison to the results of the non-minimal network in Figures 8.10 and B.41, B.42 are the following.

1. The observations for the non-minimal single network for multiple-plant emulation also hold for the minimal realization case.

2. There is no degradation in performance due to the less complex network achieved by the minimal realization procedure.

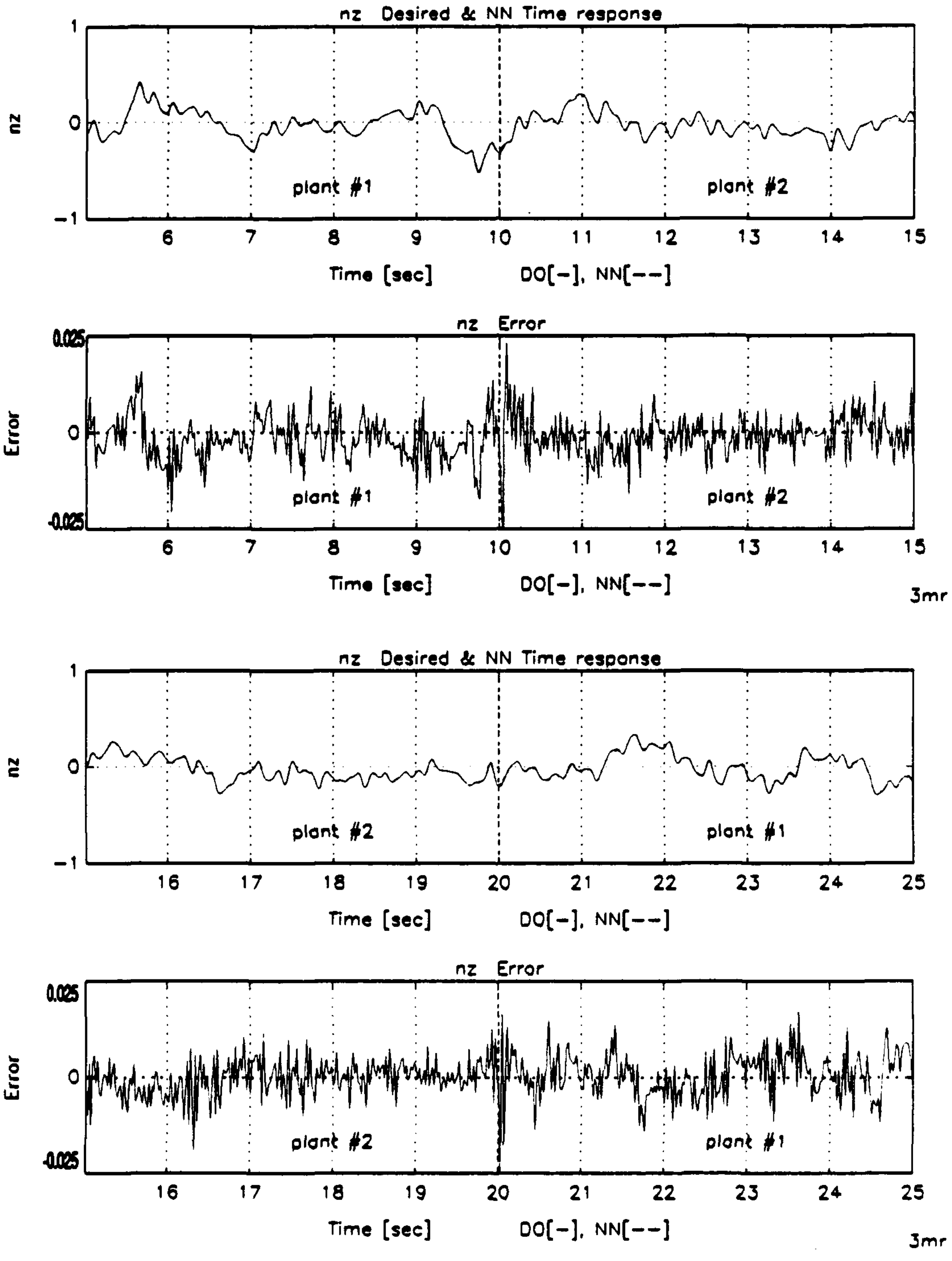


Figure 8.18: **Normal Acceleration Response of a Minimally Realized Single Network Emulating Two Plants Due to RBS Excitation of All Inputs**

D. EMULATION ANALYSIS

The capabilities of an artificial neural network to emulate MIMO dynamical systems have been shown. The main results are the following.

1. A combination of feedforward and recurrent networks is used to emulate MIMO dynamical systems.

2. The network structure is selected to match the class that the system belongs to.

3. A single network is capable of emulating multiple plants under proper training scheme.

4. More presentations are needed to emulate multiple plants.

5. A minimal realization of a network can be found, with no degradation in performance. Corollary: there is no unique solution to the emulation / identification problem.

E. ADAPTIVE CONTROL OF MIMO DYNAMICAL SYSTEM

The control results include two parts. Initially a model follower controller is designed for each of the undamaged and damaged aircraft separately. Then the two controllers are combined into a single network. The unified network is an adaptive controller. Based on the input information, the network generates the required control actions such that the plant output will behave similar to the reference model. The input information includes the pilot command p_x, the plant output feedback, and prior measurements of plant input (network output) and output as indicators of plant type, of either undamaged or damaged aircraft.

1. Model-Reference Controller - Frequency Domain Results and Discussion

The spectral response of the pitch rate q, the normal acceleration n_z and the angle of attack *AoA* of the undamaged (plant #1) and damaged (plant #2) aircraft due to pilot commands p_x is given in Figure 8.19. Each graph shows the analytic frequency response - the solid line, which is the Bode plot of the reference system outputs due to the input, together with the desired values and network output values, both analyzed using FFT, while exciting the input (pilot) command with random binary sequence (RBS). The desired value shown by the dashed line is calculated using the reference model, and the dash-dot line is the neural network output. Figure 8.20 presents the same information using the unified controller, which is capable of controlling both the undamaged and damaged plants simultaneously. The normalization of the models to ± 1 cause the shift of about -15dB from the zero dB line.

The observations from the spectral results in Figures 8.19 and 8.20 are the following.

1. Each plant is controlled such that the overall system behave very close to the reference model over the entire spectral range of interest. The difference at the low frequencies is a problem of persistent excitation. For piloted aircraft control usually the indicated difference at the low end does not present a control problem. The human pilot compensate for the low frequency dynamics with no accessional control effort.

2. The spectral behavior of the unified controller in Figure 8.20 is similar for both models, with good agreement to the analytical reference, although less accurate than the single controller shown in Figure 8.19. The difference between the unified and single controllers is due to the greater complexity of the unified controller.

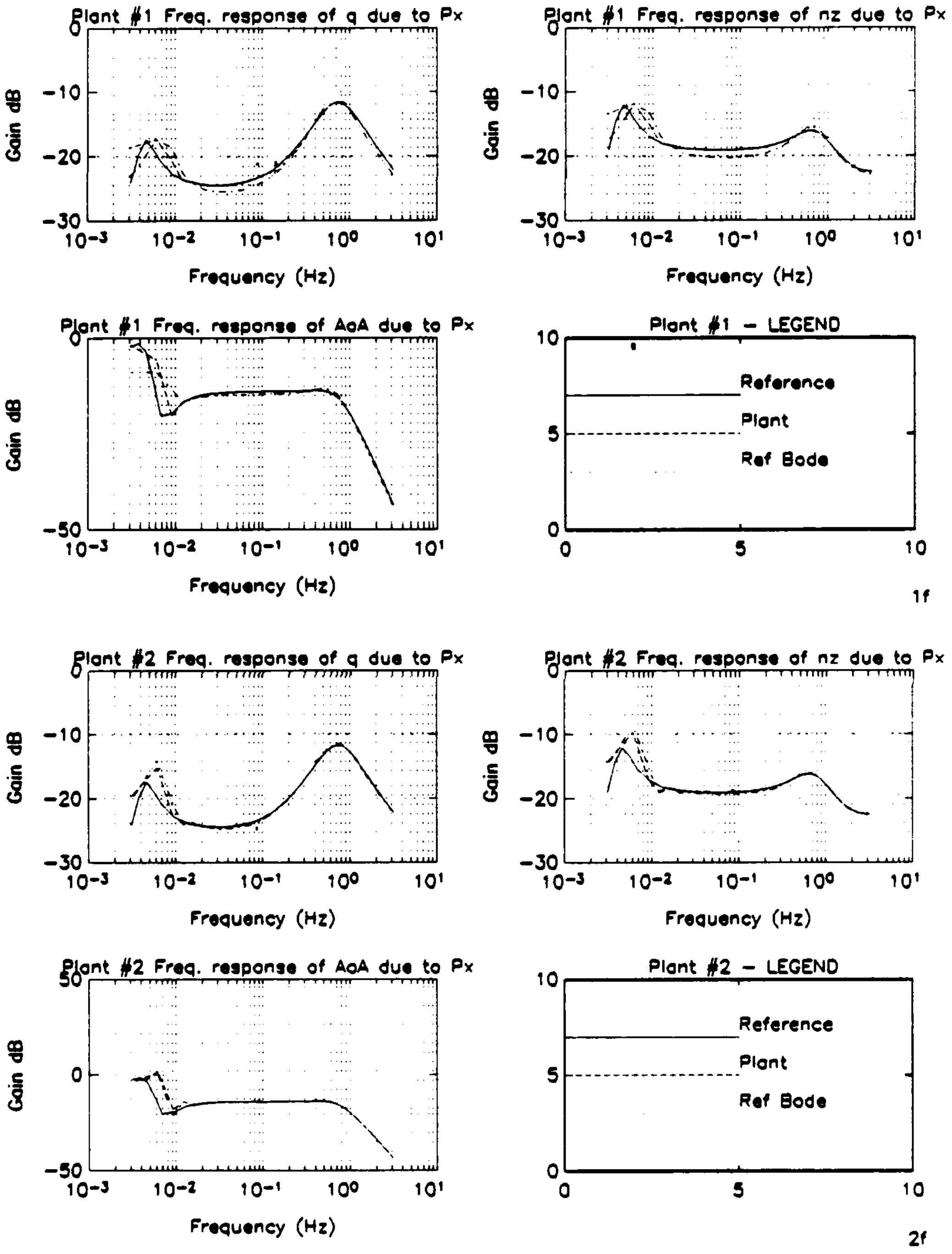


Figure 8.19: **Spectral Response of Undamaged and Damaged Controlled Aircraft Due to Pilot Longitudinal Stick Command, Single Controller**

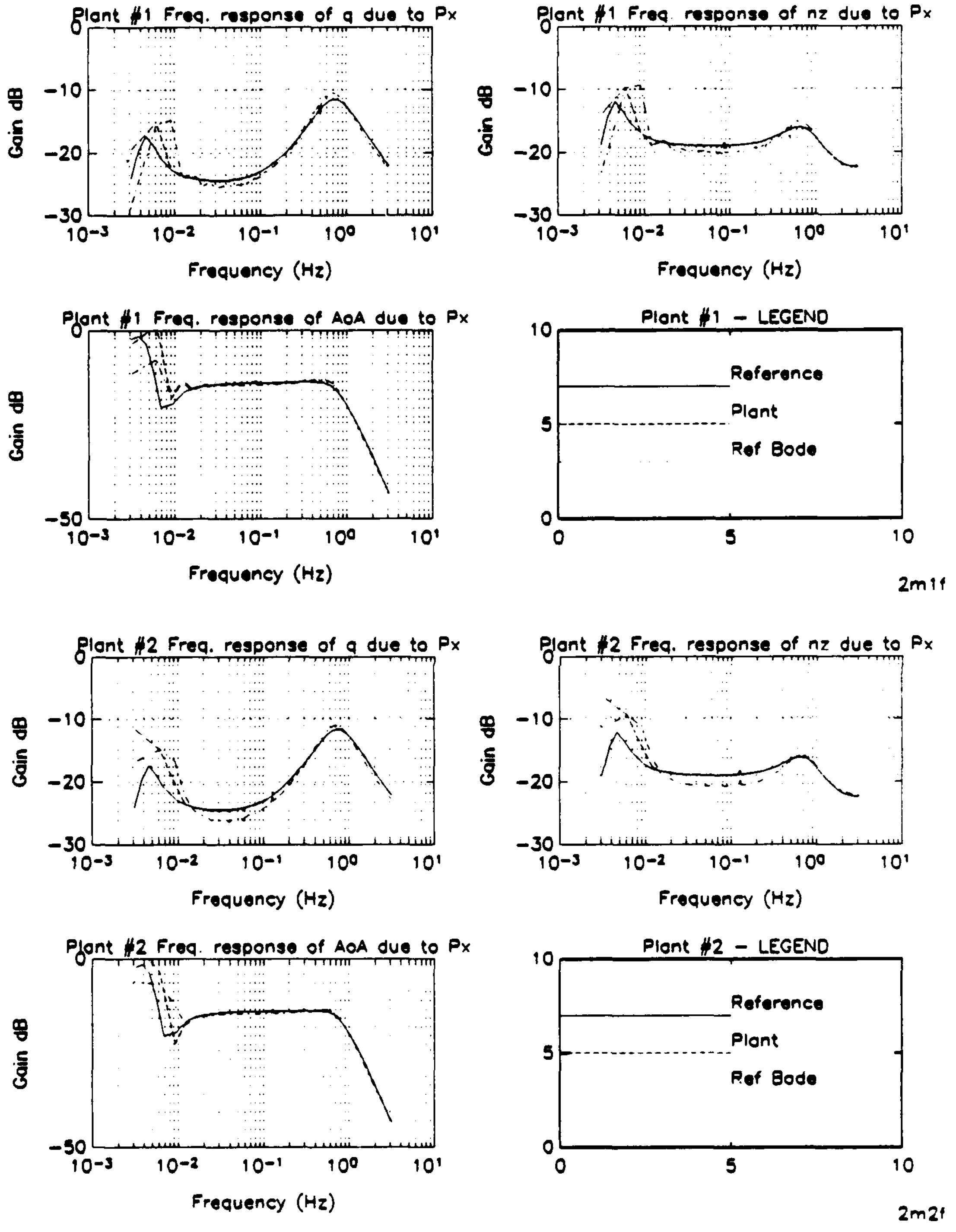


Figure 8.20: **Spectral Response of Undamaged and Damaged Controlled Aircraft Due to Pilot Longitudinal Stick Command, Unified Controller**

2. Model-Reference Controller - Time Domain Results and Discussion

The time domain analysis shows the response of the outputs of the system to a dipole test signal at the pilot command p_x. The dipole presents a one second full stick movement that demand a $5g$ acceleration from the aircraft, in both directions. Each graph shows the reference, which is the desired output, and the network output q, n_z and α for the dipole command. The error between the two is presented in the lower plot of each pair. Figure 8.22 presents the same information on the damaged aircraft, each controlled with a separate neural network controller. The stabilators, leading and trailing edge flaps actuators commands in response to a dipole input are shown in Figure 8.23 for the undamaged aircraft, and in Figure 8.24 for the damaged aircraft.

The performance of the unified controller in the time domain is tested by presenting a full stick RBS input as the pilot command. The response of the network compared to the reference model is presented in Figure 8.25 with the corresponding actuators commands in Figure 8.26.

Each graph shows the desired and network output for the command. The error between the two is presented in the lower plot of each pair. Note that the output errors are at the order of 10^{-2} or less. The relative error of all input-output relations is about 1-5%.

Examining the results shown on Figures 8.22 - 8.26 as a complete set describing the neural network controlled undamaged and damaged aircraft, brings the following general observations.

1. A neural network based controller was designed for the undamaged as well as the damaged aircraft.

2. The network finds a suitable solution, although not unique. In the undamaged aircraft the three control pairs are symmetric. Depending on the initial weights randomization the network is trained to find an equilibrium solution which is not necessarily symmetric as one could expect in Figure 8.23.

3. The network is clearly able to identify the damage in the second plant and disregard the malfunctioning surface (the left stabilator command is practically zero). The neural network learns to compensate for the loss of control surface with the rest of the actuators, mainly with the adjacent right stabilator as can be clearly seen in Figures 8.24 and 8.26.

4. The unified controller is able to identify the change in the plant and switch over to generate the required control commands. The physical output variables remain continuous, while the behavior of the control surfaces changes *dramatically*.

3. Control Analysis

In this section the capabilities of an artificial neural network to adaptively control a time-varying MIMO dynamical systems have been shown. The main results are the following.

1. An artificial neural network was trained as controller to drive a plant in a model following scheme.

2. The neural network controller was able to compensate for a loss of control action.

3. A single network is capable of controlling multiple plants in an adaptive fashion.

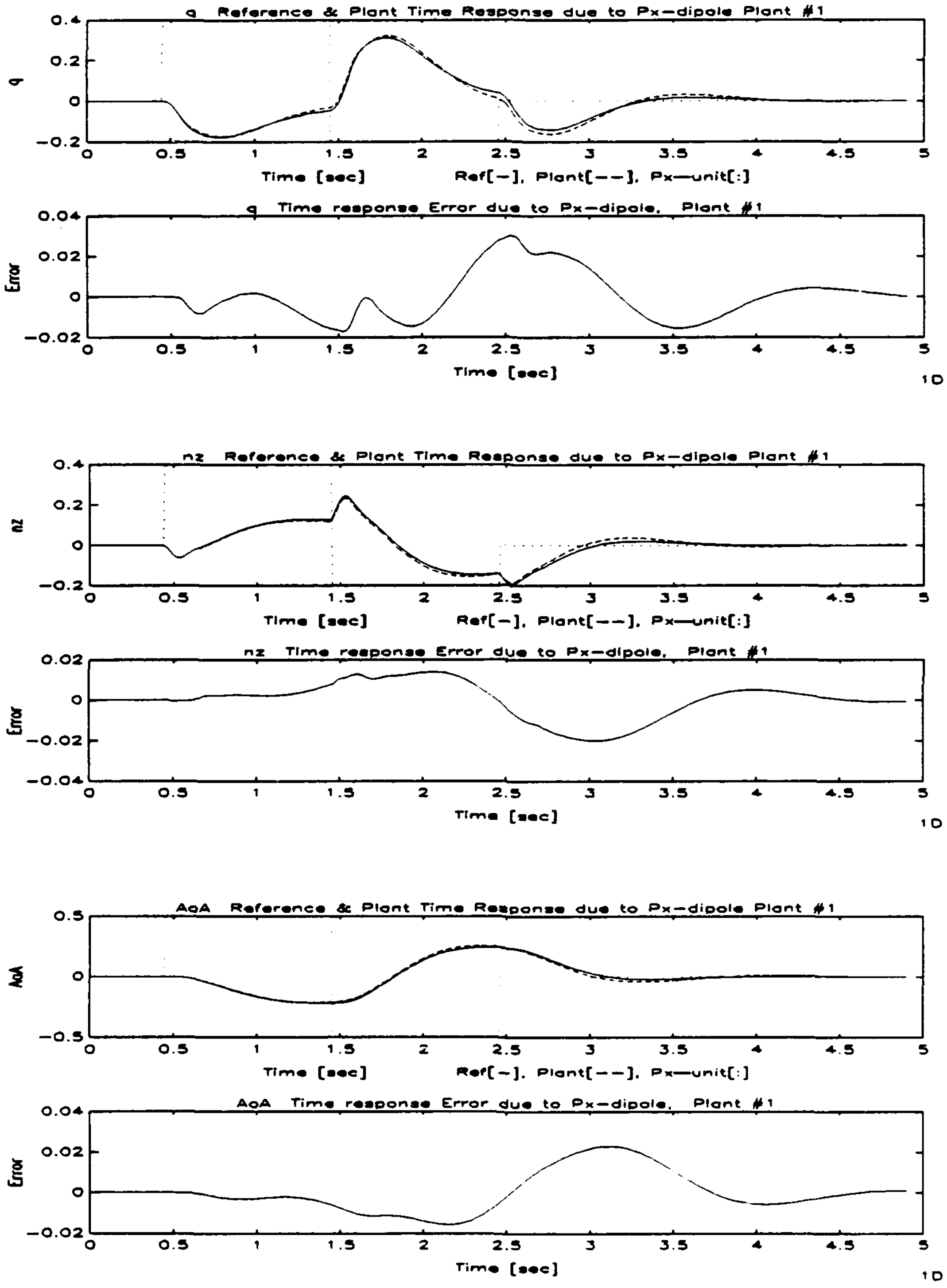


Figure 8.21: Time Response of Undamaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller

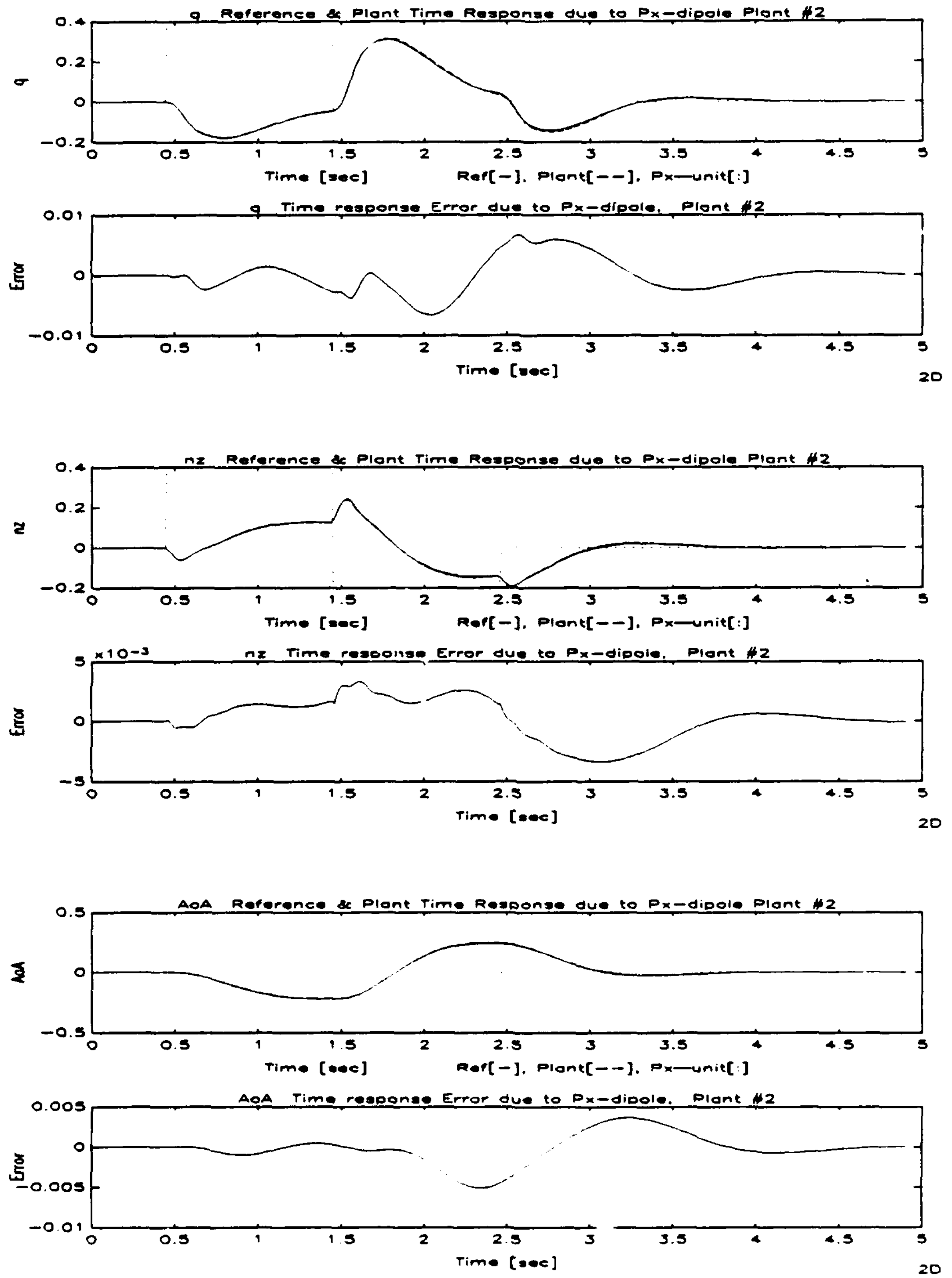


Figure 8.22: **Time Response of Damaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller**

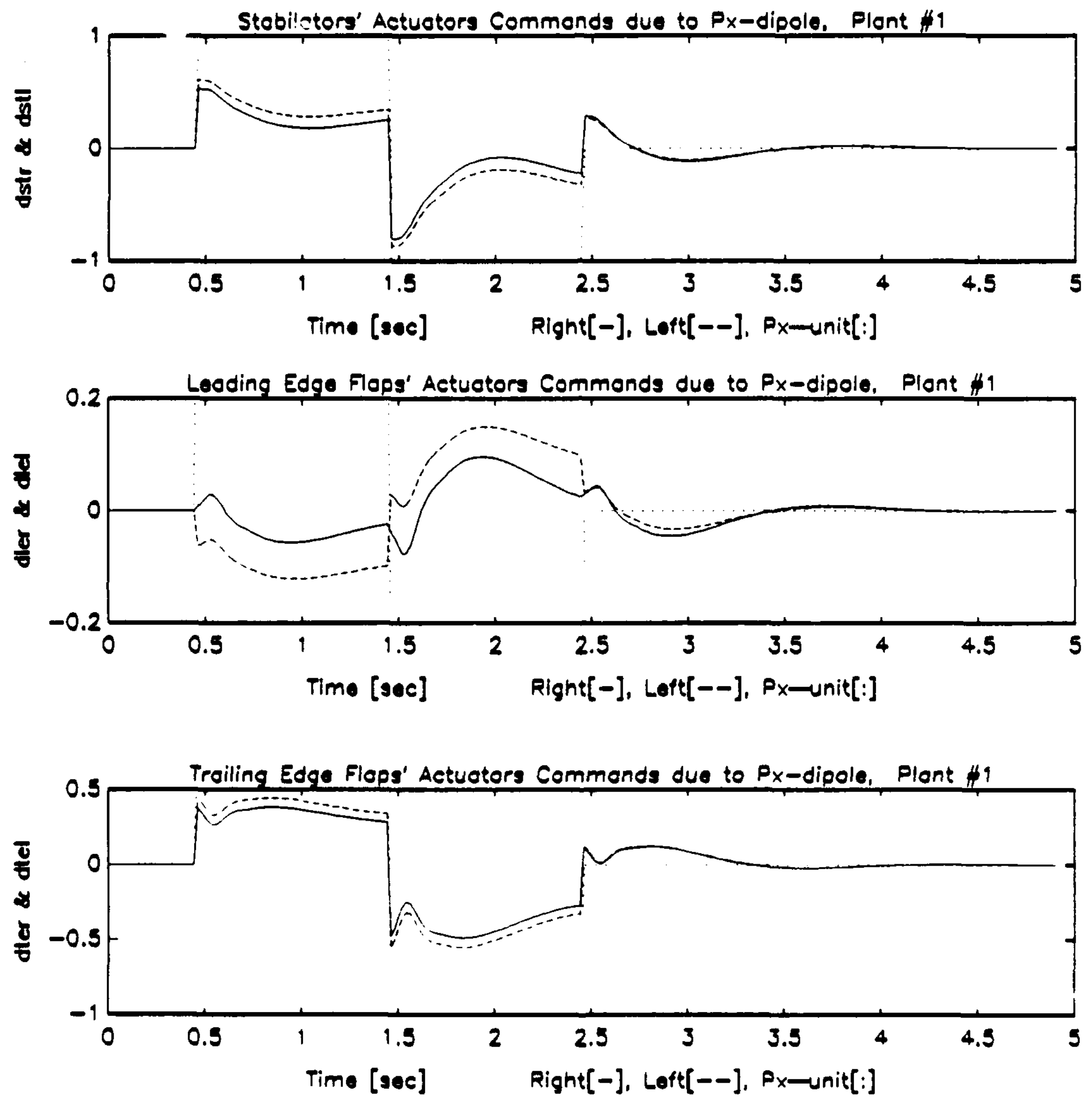


Figure 8.23: Actuation Commands of Undamaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller

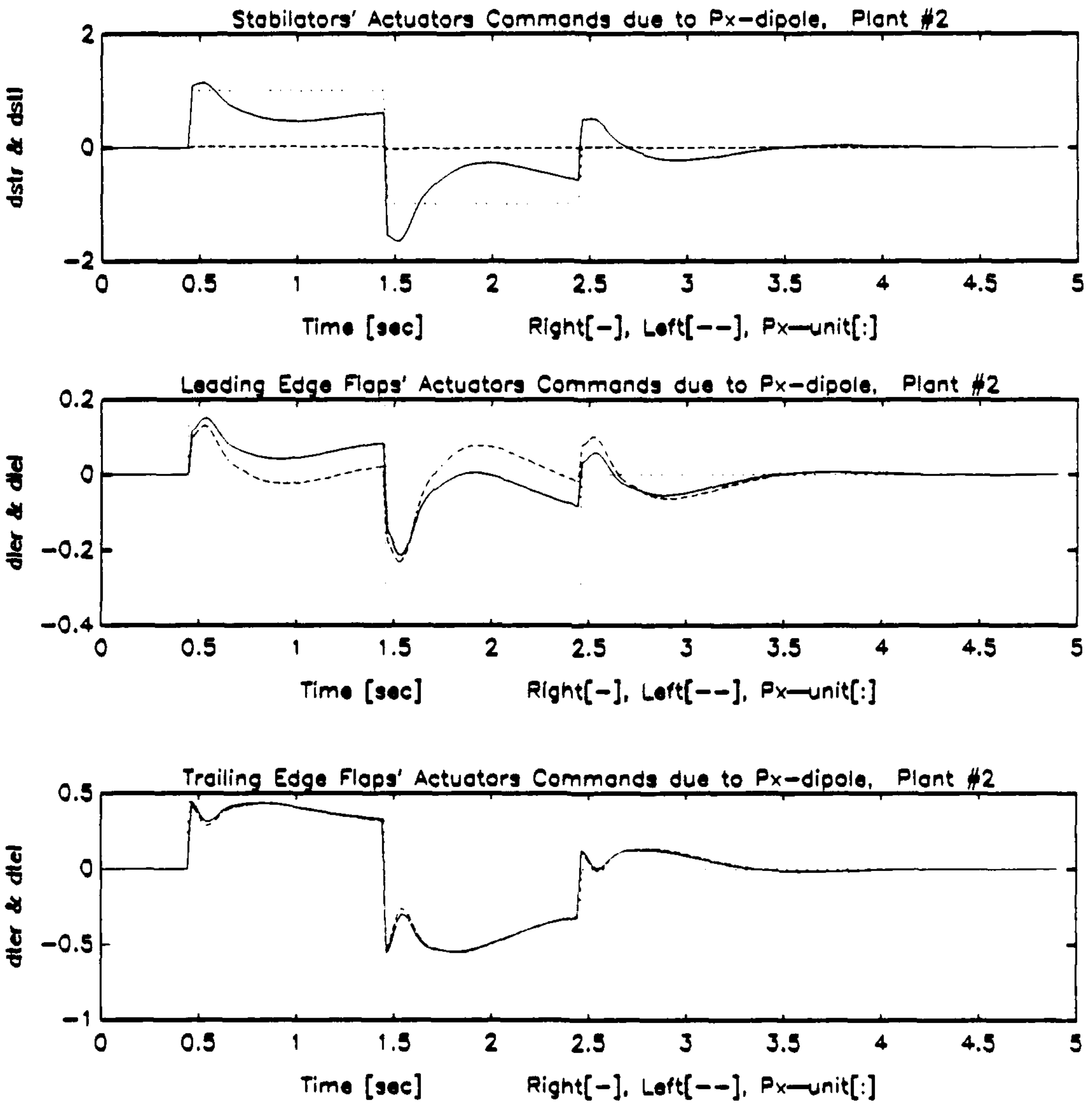
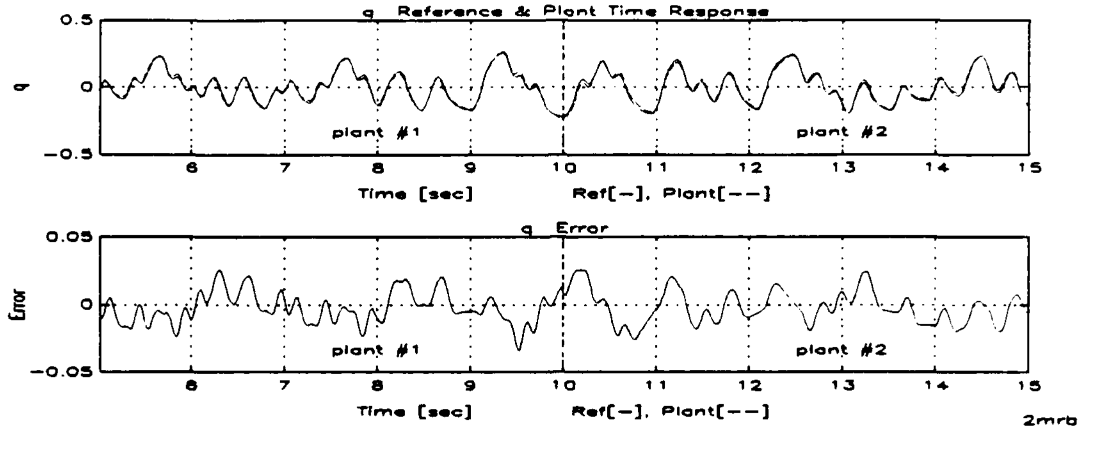
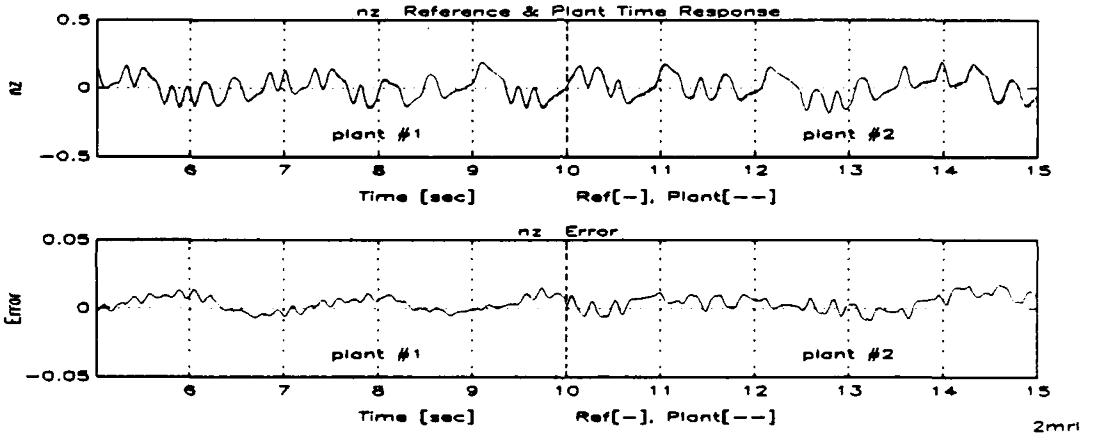


Figure 8.24: **Actuation Commands of Damaged Controlled Aircraft Due to Pilot Dipole Stick Command, Single Controller**





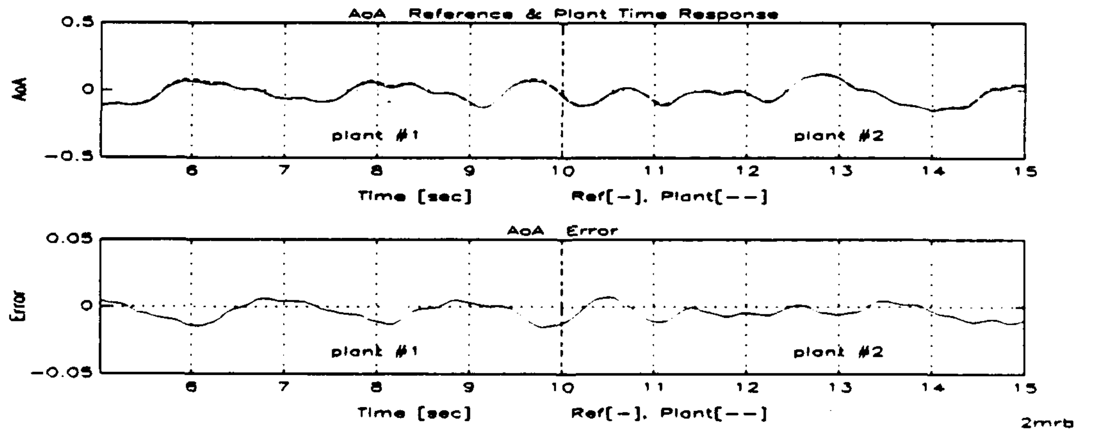


Figure 8.25: Time response of Controlled Time Varying Aircraft Due to Pilot RBS Stick Excitation, Unified Controller

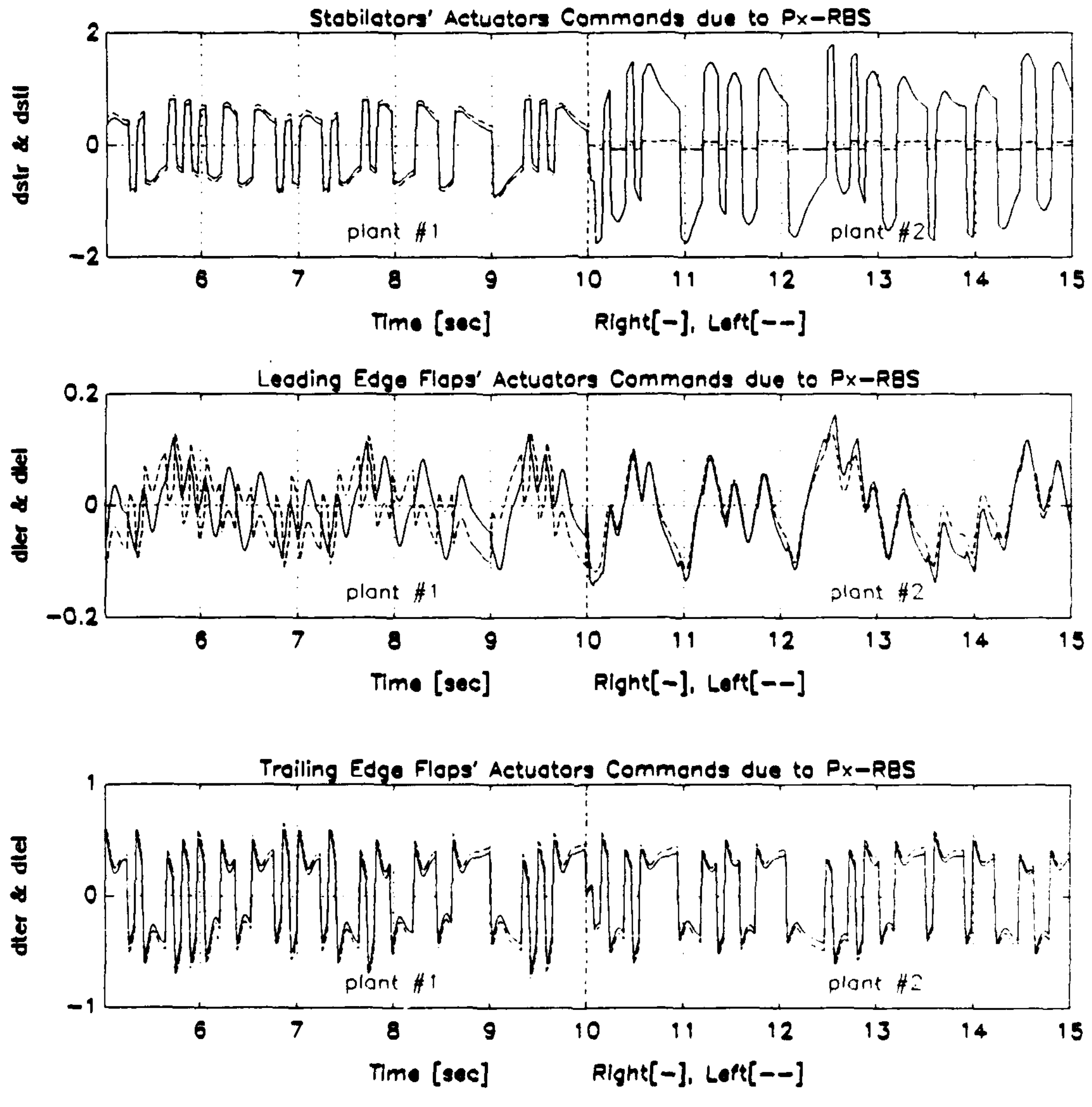


Figure 8.26: **Actuation Commands of Controlled Time Varying Aircraft Due to Pilot RBS Stick Excitation, Unified Controller**

IX. CONCLUSIONS AND RECOMMENDATIONS

This work considered dynamical systems in association with artificial neural networks. The basic theory of neural network was presented. Methods for emulation and control of dynamical systems using neural networks were developed. The important results are summarized here and some recommendations for further research are highlighted.

A. MAIN RESULTS

This research established a general approach for identification and emulation of non-linear MIMO time-varying dynamical system. Four general MIMO dynamical models are presented. Based on the selected model a corresponding network structure was introduced. The structure associated a combination of feedforward and recurrent artificial neural networks. A serial-parallel training scheme was used to train the network. A single network was shown to be capable of representing multiple dynamical systems from the same class, under the balanced random training procedure. The model switching in the procedure did not affect the accuracy of the network.

An algorithm to find minimal realization of a neural network was also presented. It was shown that there was no loss in performance of the reduced neural network, comparing to a more complex network that does the same task.

Further in the research a neural network was trained to control a MIMO dynamical system using a back-propagation approach through the plant training procedure, which is a variation of the basic back-propagation algorithm. The network was trained in a model-follower control architecture without a direct access to the desired values and without the need for specific knowledge on the controlled system. A single network controller was trained to represent multiple controllers, thus forming an adaptive controller. No estimation process was involved in the procedure.

Artificial neural networks were found to be a powerful tool for system identification and in designing adaptive controllers.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

Several topics that were addressed in the research need to be further investigated. The issues are discussed below with possible directions of approach.

In the minimal realization procedure, singular values of a weight matrix were involved. In the LTI system emulation the singular values had physical meaning. It seems that the singular values may give more information about the system. It can also serve as an indicator of network training progress and also as a feature detector. The singular value analysis of the weight matrix needs more investigation in the above directions as well as in improving the minimization procedure to more definite results.

Training recurrent networks has still many open aspects. Since recurrent networks are vital in association with dynamical system, more research is needed in the theoretical and practical aspects of the training. Methods for stable algorithms of training hidden recurrent layers are affiliated with this direction of research as well. General constraints on the learning rate for stable training are greatly desired. Proper learning rate may also improve convergence rate. Methods involving numerical analysis in stability theory may be an approach.

Algorithms for utilizing adaptive neural networks controller in real time and on-line applications may be seen as a direct extension of this work. This area of research may fined many current applications, especially in the non-linear regime.

APPENDIX A. F/A-18A MATHEMATICAL MODEL

The F/A-18A mathematical model for the damaged and undamaged configurations is given this MATLAB™ script file. The aerodynamical stability derivatives are shown in the body of the program.

```
% f18longN.m
%
%  This file calculates the continuous state space representation of the
% longitudinal modes of the f18 (with or without malfunctions)
% fighter aircraft.
% Then the matrices are balanced and discretized according to a given
% sampling interval - ts. The results are to be loaded to file:
% f18long_model.h
%    The state vector:   x=[u(ft/sec),w(ft/sec),q(rad/sec),theta(rad)]'
%    The control vector: del=[dstr,dstl,dler,dlel,dter,dtel]' (deg)
%    The calculation assumes that the gusts are negligible.
% Trim AOA=alpha0=arctan(W/U), and theta0=gamma0+alpha0.
% This uses cscaleb.m
cs=[];
%directory='/fil669/home/dror/f18/Emul/';
directory='/fil669/home/dror/f18/Cntrl/';
ts=0.02;                    % [sec]- Sampling interval
alpha0=2.6184*pi/180;       % [rad] - Trim angle of attack
gamma0=0.0;                 % [rad] - Path angle
theta0=alpha0+gamma0;       % [rad] - a/c body angle
U=646.42;                   % [ft/sec] - True airspeed
alt=10000;                  % [ft] - altitude
M=0.6;                      % [#] - Mach number
g=32.174;                   % [ft/sec^2] - gravity constant
W=U*tan(alpha0);
W=0;                        % In trim axes
%theta0=gamma0;             % In trim axes

disp(' Definitions of dimensional stability derivatives')

% Name       Value        Units            Partial deriv. of   With respect to
%

xu=    -0.13257e-1;       %[1/sec]         Longitudinal force Forward velocity
xw=     0.71265e-1;       %[1/sec]         Longitudinal force Vertical velocity
xq=     0.0;              %[ft/(rad-sec)]  Longitudinal force Pitch rate
xwd=    0.0;              %[--]            Longitudinal force Vertical acceler
xdsb=   0.0;              %(ft/sec^2)/rad  Longitudinal force Speed brake
xdth=     0.14257;        %[pct/sec^2]     Longitudinal force Throttle
xds=    0.0;              %(ft/sec^2)/rad  Longitudinal force Horiz stabilator
xdlf=   0.0;              %(ft/sec^2)/rad  Longitudinal force Lead edge flap
xdtf=   0.0;              %(ft/sec^2)/rad  Longitudinal force Trail edge flap
zu=    -0.73337e-1;       %[1/sec]         Vertical force     Forward velocity
zw=    -1.1526;           %[1/sec]         Vertical force     Vertical velocity
zq=    -5.6525;           %[ft/(rad-sec)]  Vertical force     Pitch rate
zwd=  -0.006917;          %[--]            Vertical force     Vertical acceler
zdsb=   0.0;              %(ft/sec^2)/rad  Vertical force     Speed brake
zdth=   0.0;              %[pct/sec^2]     Vertical force     Throttle
zds=   -129.5;            %(ft/sec^2)/rad  Vertical force     Horiz stabilator
zdlf=   19.43;            %(ft/sec^2)/rad  Vertical force     Lead edge flap
```

```
zdtf=  -149.0;            %(ft/sec^2)/rad Vertical force     Trail edge flap
mu=    -0.12988e-4;       %[1/sec-ft]     Pitching moment    Forward velocity
mw=    -0.11331e-1;       %[1/sec-ft]     Pitching moment    Vertical velocity
mq=    -0.59346;          %[1/sec]        Pitching moment    Pitch rate
mwd= -0.34049e-3;         %[1/ft]         Pitching moment    Vertical acceler
mdsb=   0.0;              %(1/sec^2)/rad  Pitching moment    Speed brake
mdth=   0.0;              %[pct/(sec^2-ft)]Pitching moment   Throttle
mds=   -15.6;             %(1/sec^2)/rad  Pitching moment    Horiz stabilator
mdlf=  -1.609;            %(1/sec^2)/rad  Pitching moment    Lead edge flap
mdtf=   1.499;            %(1/sec^2)/rad  Pitching moment    Trail edge flap
%-----------------------------------------------------------------------

model=input('ENTER MODEL # ','s');

%-----------------------------------------------------------------------
disp('actuators transfer functions, and their state space representation')
% ======================================================================
% Actuators are scaled to +/-1

% STABILATOR
stln=[1/(82.9)^2, 2*(0.068)/(82.9), 1];% left side numerator
stld=conv([1/(36.4)^2,2*(0.41)/(36.4),1],[1/(105.3)^2,2*(0.59)/105.3,1]);
mxstl=10.5;          % maximum positive deflection of left stabilator [deg]
mnstl=-24;           % minimum negative deflection of left stabilator [deg]
strn=stln;           % right side numerator
strd=stld;           % right side denominator
mxstr=10.5;          % maximum positive deflection of right stabilator [deg]
mnstr=-24;           % minimum negative deflection of right stabilator [deg]
if(model=='2') stln=0; end;
[Fstl,Gstl,Hstl,Dstl]=tf2ss(stln,stld);
[Fstr,Gstr,Hstr,Dstr]=tf2ss(strn,strd);
nustl=(mxstl-mnstl)/2;            % Left stabilator scaler
bistl=(mxstl+mnstl)/(2*nustl);    % Left stabilator bias
nustr=(mxstr-mnstr)/2;            % Right stabilator scaler
bistr=(mxstr+mnstr)/(2*nustl);    % Right stabilator bias

% LEADING EDGE FLAP
leln=1;
leld=conv([1/(26.9), 1],[1/(82.9), 1]);
mxlel=34;            % maximum positive deflection - left leading edge flap [deg]
mnlel=-3;            % minimum negative deflection - left leading edge flap [deg]
lern=leln;
lerd=leld;
mxler=34;            % maximum positive deflection of right leading edge flap [deg]
mnler=-3;            % minimum negative deflection of right leading edge flap [deg]
[Flel,Glel,Hlel,Dlel]=tf2ss(leln,leld);
[Fler,Gler,Hler,Dler]=tf2ss(lern,lerd);
nulel=(mxlel-mnlel)/2;            % Left leading edge flap scaler
bilel=(mxlel+mnlel)/(2*nulel);    % Left leading edge flap bias
nuler=(mxler-mnler)/2;            % Right leading edge flap scaler
biler=(mxler+mnler)/(2*nulel);    % Right leading edge flap bias

% TRAILING EDGE FLAP
teln=1;
teld=[1/(35)^2, 2*(0.71)/(35), 1];
mxtel=45;            % maximum positive deflection -left trailing edge flap [deg]
mntel=-8;            % minimum negative deflection -left trailing edge flap [deg]
tern=teln;
terd=teld;
mxter=45;            % maximum positive deflection -right trailing edge flap[deg]
mnter=-8;            % minimum negative deflection -right trailing edge flap[deg]
[Ftel,Gtel,Htel,Dtel]=tf2ss(teln,teld);
[Fter,Gter,Hter,Dter]=tf2ss(tern,terd);
nutel=(mxtel-mntel)/2;            % Left trailing edge flap scaler
bitel=(mxtel+mntel)/(2*nutel);    % Left trailing edge flap bias
nuter=(mxter-mnter)/2;            % Right trailing edge flap scaler
biter=(mxter+mnter)/(2*nutel);    % Right trailing edge flap bias
```

```
% Scaling and biasing
NU=[nustl nustr nulel nuler nutel nuter];  % scale input from+/-1 to physical
bi=[bistl bistr bilel biler bitel biter];  % input shift (bias)
NU=max(abs([[mxstl;mnstl],[mxstr;mnstr],[mxlel;mnlel],[mxler;mnler],
[mxtel;mntel],[mxter;mnter]]));
bi=zeros(bi);
disp(' sensors transfer functions, and their state space representation')
% ======================================================================

% RATE GYRO
rgqn=[1/(131.7), 1];                          % Pitch rate rate-gyro
rgqd=conv([1/(167.8), 1],[1/(461.7), 1]);
[Frgq,Grgq,Hrgq,Drgq]=tf2ss(rgqn,rgqd);

% ACCELEROMETER
aczn=[1/(235.8), 1];                          % Normal accelerometer
aczd=[1/(395.3)^2, 2*(0.96)/(395.3), 1];
[Facz,Gacz,Hacz,Dacz]=tf2ss(aczn,aczd);

% ANGLE OF ATTACK
aan=1;
aad=[1/(14), 1];
[Faa,Gaa,Haa,Daa]=tf2ss(aan,aad);

%---------------------------------------------------------------------

disp(' Calculation of longitudinal matrices')
% calculating continuous longitudinal dynamic matrix.
% In:Xx=[u,w,q,theta]', Out:Xxd

Fx(2,:)=[zu zw (zq+U) -g*sin(theta0)]/(1-zwd);
Fx(1,:)=[xu xw (xq-W) -g*cos(theta0)]+xwd*Fx(2,:);
Fx(3,:)=[mu mw   mq      0          ]+mwd*Fx(2,:);
Fx(4,:)=[0, 0,   1,      0          ];

% calculating continuous longitudinal control matrix.
% In:Dx=[dstx,dlex,dtex]', Out:Xxd

Gx(2,:)=[zds zdlf zdtf]/(1-zwd);
Gx(1,:)=[xds xdlf xdtf]+Gx(2,:)*xwd;
Gx(3,:)=[mds mdlf mdtf]+Gx(2,:)*mwd;
Gx(4,:)=[0,  0,   0  ];

% calculating longitudinal measurement matrix. In:Xx, Out:Yx=[q,nz,alpha]'

Hx(1,:)=[0, 0,   1, 0];
Hx(2,:)=Fx(2,:)-Hx(1,:)*U; %nz=wdot-U*q=U*(alphadot-q)
Hx(3,:)=[0, 1/U, 0, 0];


% calculating longitudinal control feedforward matrix.
% In:Dx=[dstx,dlex,dtex]', Out:Yx

Dx(1,:)=[0, 0, 0];
Dx(2,:)=Gx(2,:);
Dx(3,:)=[0, 0, 0];

%---------------------------------------------------------------------

% calculating longitudinal controls distribution.
% In:D=[dstr,dstl,dler,dlel,dter,dtel]',Out:Dx=[dstx,dlex,dtex]'

lng=zeros(3,6);
lng(1,1)=0.5;lng(1,2)=0.5;lng(2,3)=0.5;lng(2,4)=0.5;lng(3,5)=0.5;lng(3,6)=0.5;

%---------------------------------------------------------------------
```

```
disp(' appending system matrices')
% system dynamic matrix. In:X=[u,w,q,theta]', Out: Xd.
% control matrix. In:D=[dstr,dstl,dler,dlel,dter,dtel]', Out:Xd
% measurement matrix. In:X, Out:Y=[q,nz,alpha]'.
% control feed-forward matrix. In:D, Out:Y .

Fm=Fx;
Gm=Gx*lng;
Hm=Hx;
Dm=Dx*lng;
%-----------------------------------------------------------------

% convert from (per radian) to (per degree) and from (ft/sec^2) to (g)
disp('convert (/rad) to (/deg) & (ft/sec^2) to (g)');
[nx,nu]=size(Gm);[ny,nu]=size(Dm);
r2d=180/pi;
d2r=1/r2d;
Nu=eye(nu)*d2r;                   % Get inputs in degrees
Nx=diag([1,U,1,1]);               % change w to alpha
Ny=diag([d2r,g,d2r]);             % convert to degrees and g's
Fm=Nx\Fm*Nx;
Gm=Nx\Gm*Nu;
Hm=Ny\Hm*Nx;
Dm=Ny\Dm*Nu;
end;

%-----------------------------------------------------------------

disp(' actuators state variable model')

[FA,GA,HA,DA]=append(Fstr,Gstr,Hstr,Dstr,Fstl,Gstl,Hstl,Dstl);
[FA,GA,HA,DA]=append(FA,GA,HA,DA,Fler,Gler,Hler,Dler);
[FA,GA,HA,DA]=append(FA,GA,HA,DA,Flel,Glel,Hlel,Dlel);
[FA,GA,HA,DA]=append(FA,GA,HA,DA,Fter,Gter,Hter,Dter);
[FA,GA,HA,DA]=append(FA,GA,HA,DA,Ftel,Gtel,Htel,Dtel);
% Scale inputs
GA=GA*diag(NU); DA=DA*diag(NU);

disp(' sensors state variable model')

[FS,GS,HS,DS]=append(Frgq,Grgq,Hrgq,Drgq,Facz,Gacz,Hacz,Dacz);
[FS,GS,HS,DS]=append(FS,GS,HS,DS,Faa,Gaa,Haa,Daa);

%-----------------------------------------------------------------
 disp('check and balance matrices')
[Fm,Gm,Hm]=ckbal('Fm','Gm','Hm',Fm,Gm,Hm);
[FA,GA,HA]=ckbal('FA','GA','HA',FA,GA,HA);
[FS,GS,HS]=ckbal('FS','GS','HS',FS,GS,HS);

%-----------------------------------------------------------------
disp('cascading the system')
[Fps,Gps,Hps,Dps]=series(FA,GA,HA,DA,Fm,Gm,Hm,Dm);
[Fps,Gps,Hps,Dps]=series(Fps,Gps,Hps,Dps,FS,GS,HS,DS);

%-----------------------------------------------------------------
% must discretize AFTER cascading continuous systems
disp('discretize the matrices')
[Fpd,Gpd]=c2d(Fps,Gps,ts);

%-----------------------------------------------------------------
% find the extrimum values of each output due to each controller and
% normalize to +/- 1.
by=zeros(1,ny);
cs=ones(1,ny);
ii=input(['Do you wish to normalize outputs to +/-1? [y]/n '],'s');
if isempty(ii), ii='y';end;
if ii=='y'   %normalize
```

```
disp('Normalizing...')
format compact
       if(model~='1') eval(['load ' directory 'zcs1.mat']);% get cs,by
       cscaleb('d',Fpd,Gpd,diag(cs)\Hps,diag(cs)\Dps,bi,by,ts,1,1);
       by,cs  % show by and cs
       else
       l=input(['Enter # of iterations [1e6]: ']);
              if isempty(l), l=1e6;end;
              while 1
              [cs,by]=cscaleb('d',Fpd,Gpd,Hps,Dps,bi,by,ts,1,l);
              i=input('Do you want to recalculate  cs, by? y/[n]  ','s');
                     if (isempty(i)) break;end;
                     if(i=='n') break;end;
              end;
       i=input('Do you want to edit  cs, by (via maxy,miny)? y/[n]  ','s');
              if (isempty(i)) i='n';end;
              if(i=='n') ;
              else
              keyboard;
              by=-0.95*(maxy+miny)./(maxy-miny)
              cs= (maxy-miny)/(2*0.95)      % The 0.95 is 5% safety factor
              end;
       format;
       end;
Hps=diag(cs)\Hps;
Dps=diag(cs)\Dps;
end;
%-----------------------------------------------------------------

i=input(['Do you want to save matrices to ' directory ' for NN use? [y]/n '],'s');
if isempty(i), i='y';end;
if i=='y'% Save
eval(['save ' directory 'zFld' model '.mat Fpd']);
eval(['save ' directory 'zGld' model '.mat Gpd']);
eval(['save ' directory 'zHld' model '.mat Hps']);
eval(['save ' directory 'zDld' model '.mat Dps']);
eval(['save ' directory 'zBil' model '.mat bi']);
eval(['save ' directory 'zBol' model '.mat by']);
eval(['save ' directory 'zcs' model '.mat cs by NU Nu Nx Ny']);
end;
format
```

APPENDIX B. TIME AND FREQUENCY RESPONSES OF ARTIFICIAL NEURAL NETWORK EMULATION OF DAMAGED AND UNDAMAGED F/A-18A AIRCRAFT

The graphs in Appendix B are referenced in chapter VIII. The reader is referred to chapter VIII for comprehensive explanation and related discussion regarding the plots that appear in this appendix.

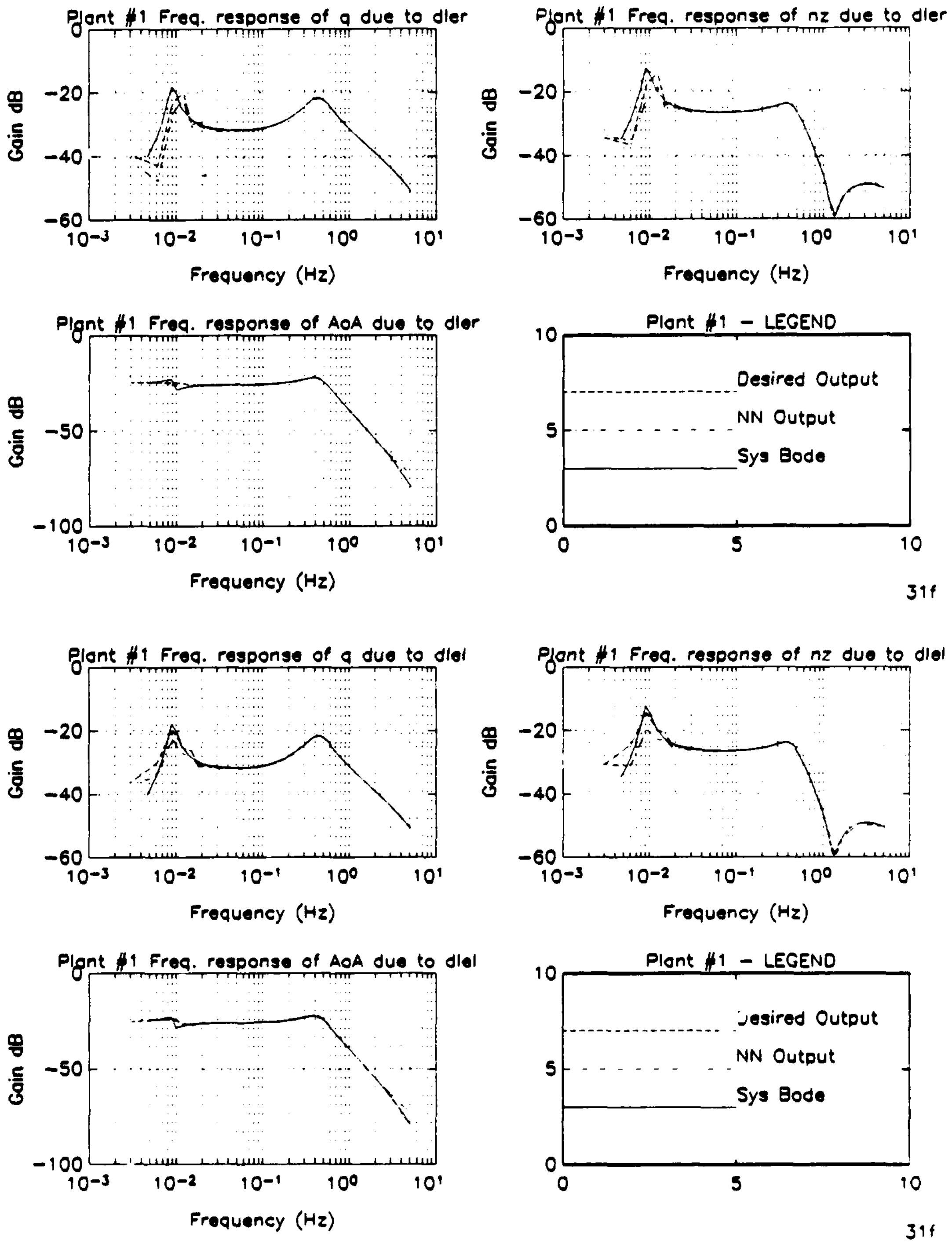


Figure B.1: **Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Single Plant, Single Network Emulation**

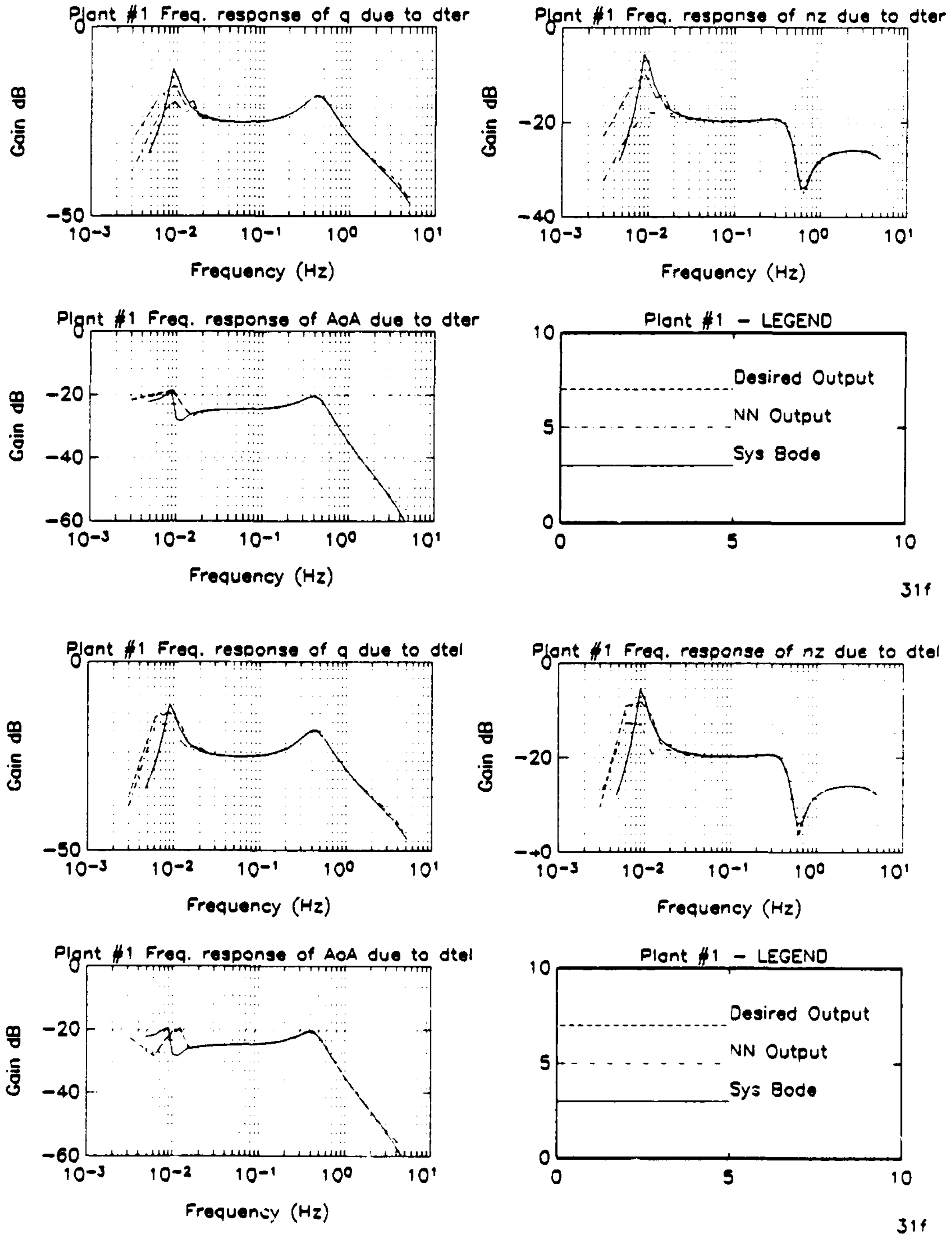


Figure B.2: **Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Single Plant, Single Network Emulation**

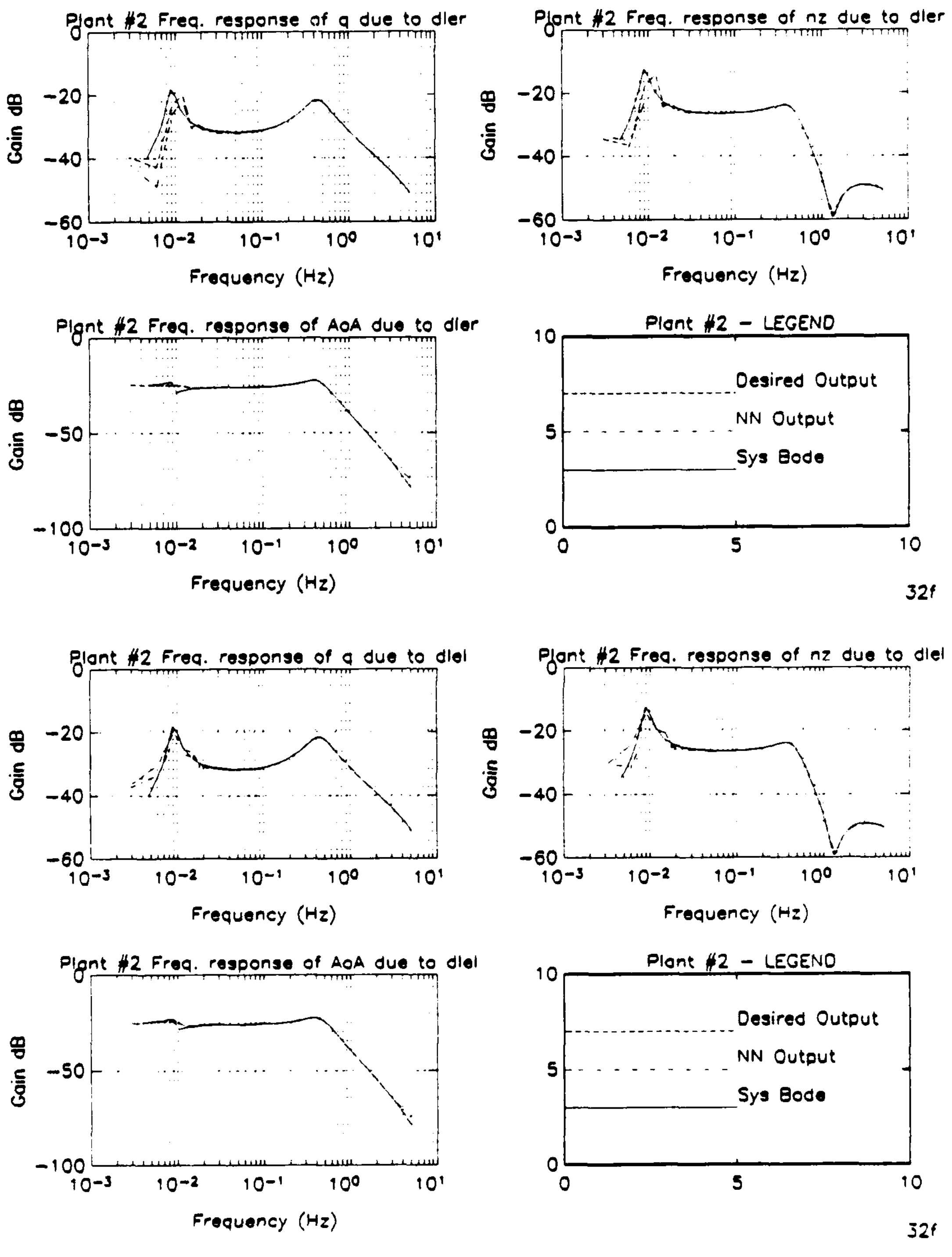


Figure B.3: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Single Plant, Single Network Emulation**

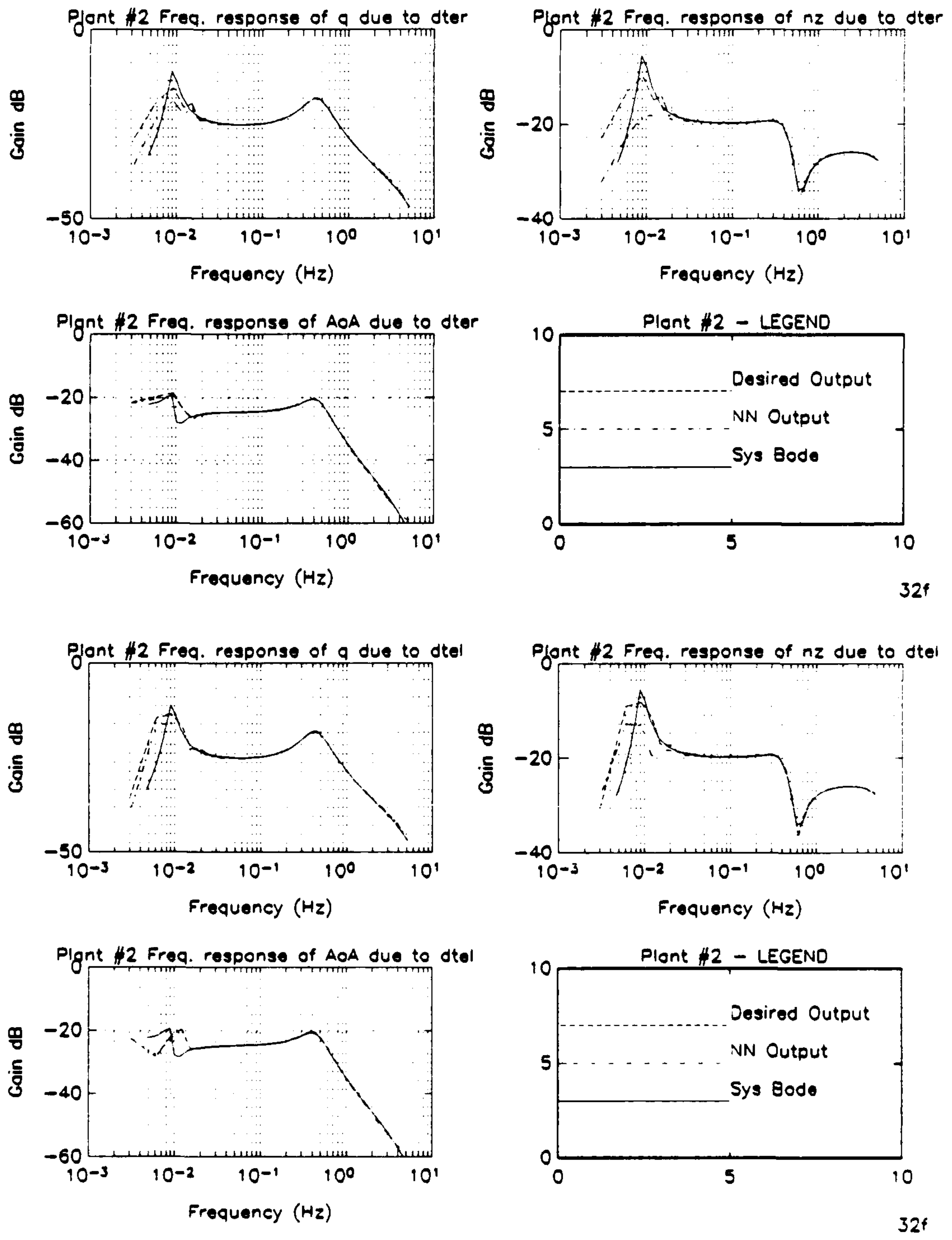


Figure B.4: **Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Single Plant, Single Network Emulation**

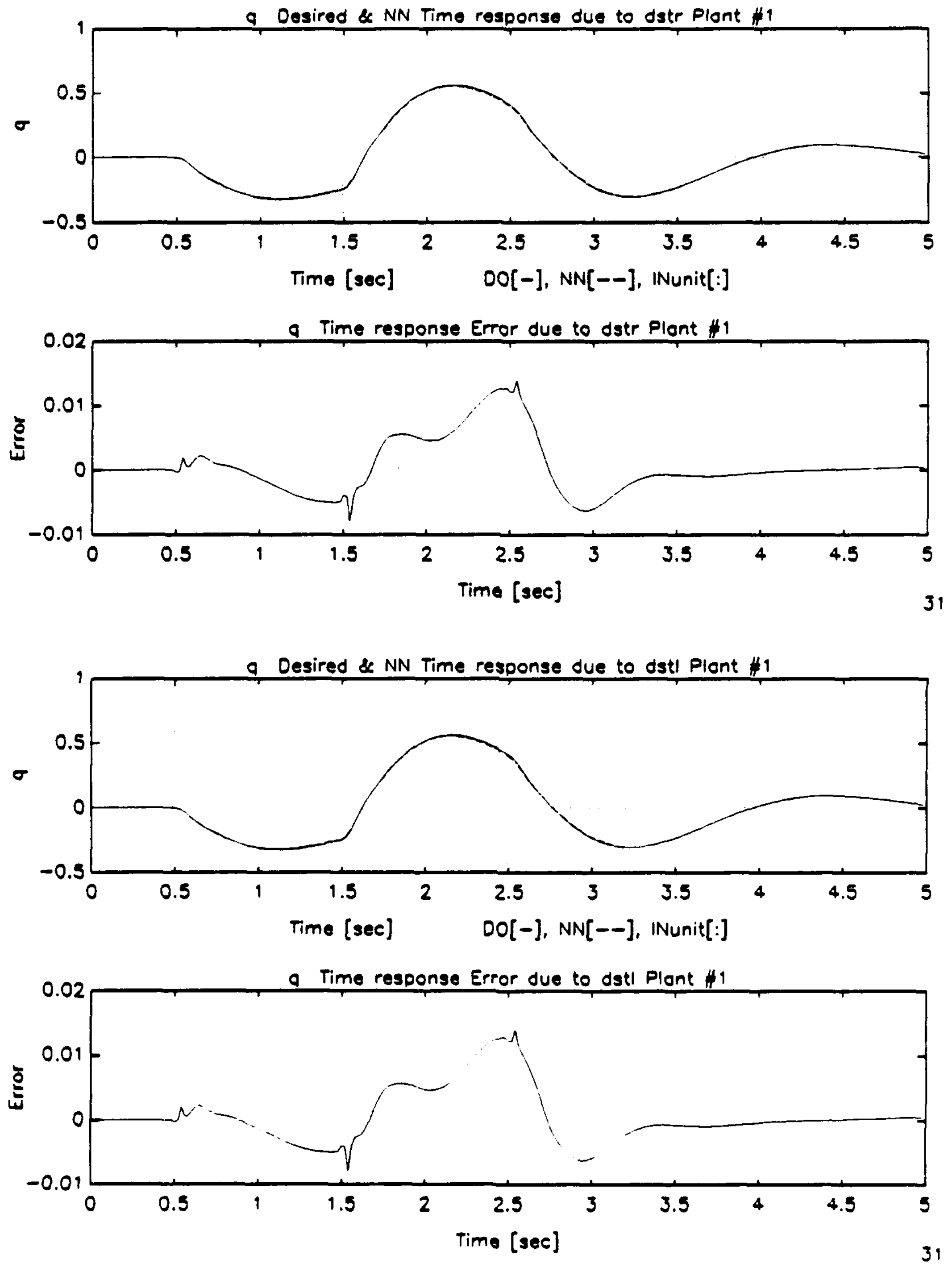


Figure B.5: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation**

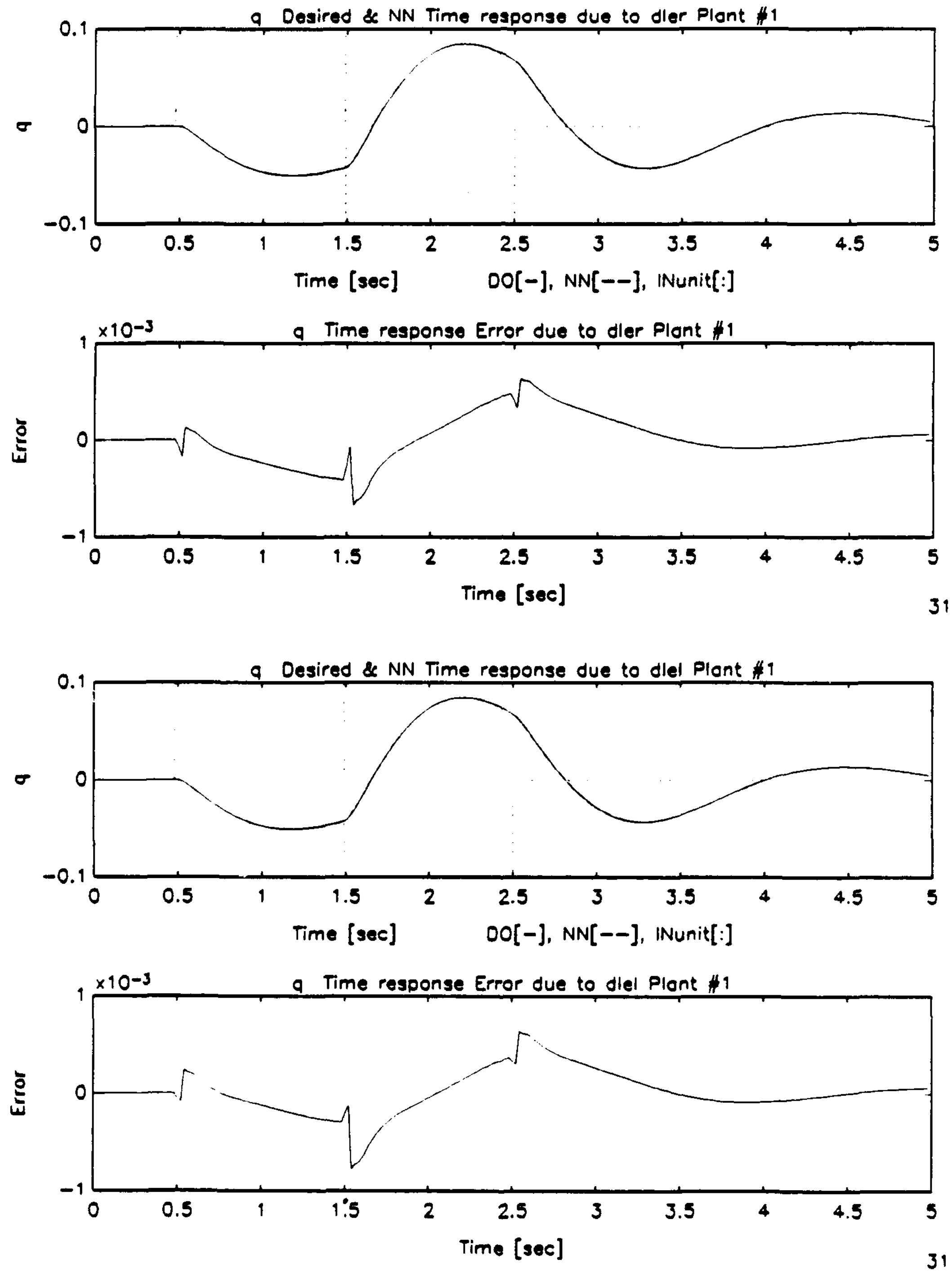


Figure B.6: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation**

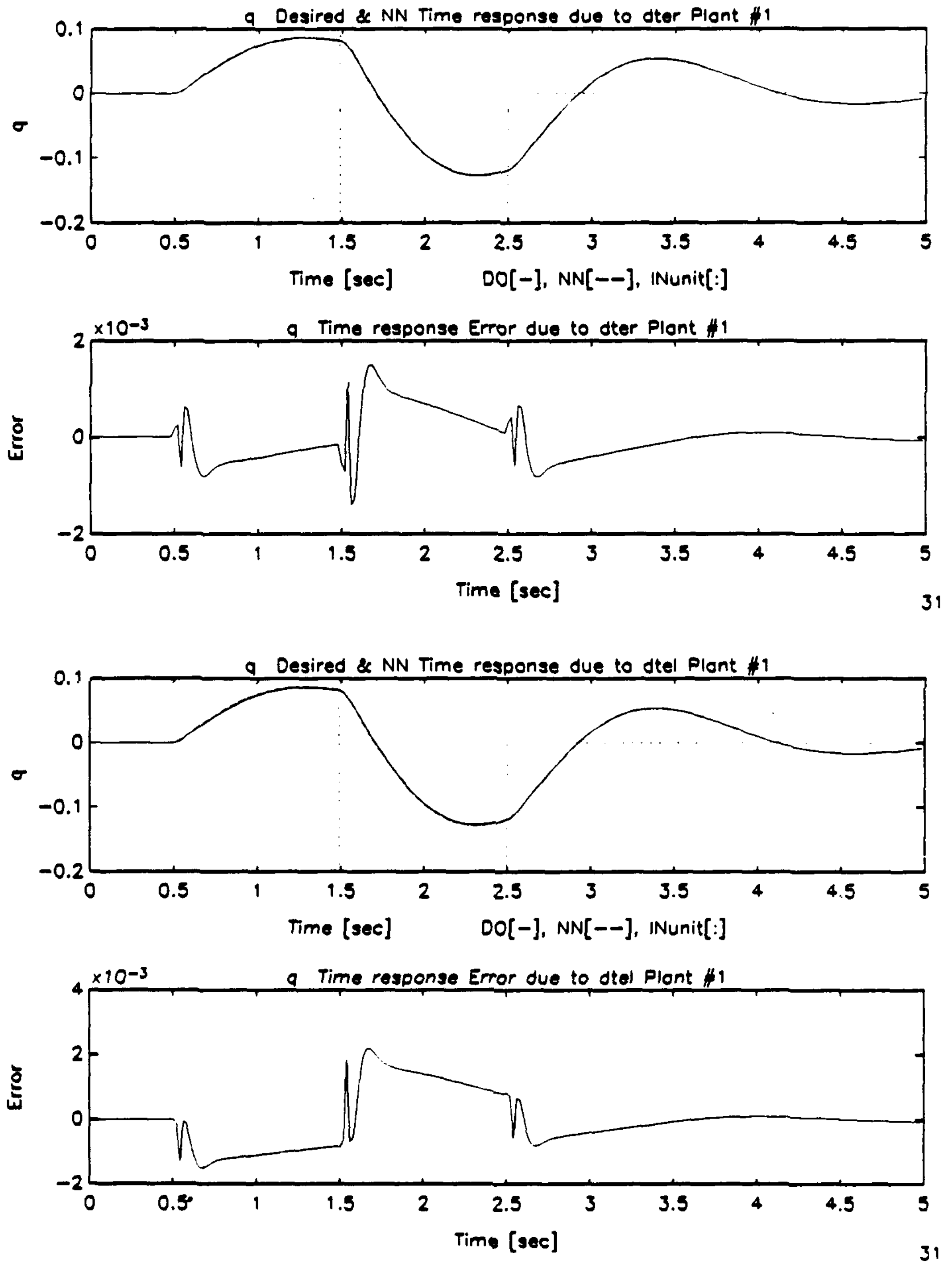
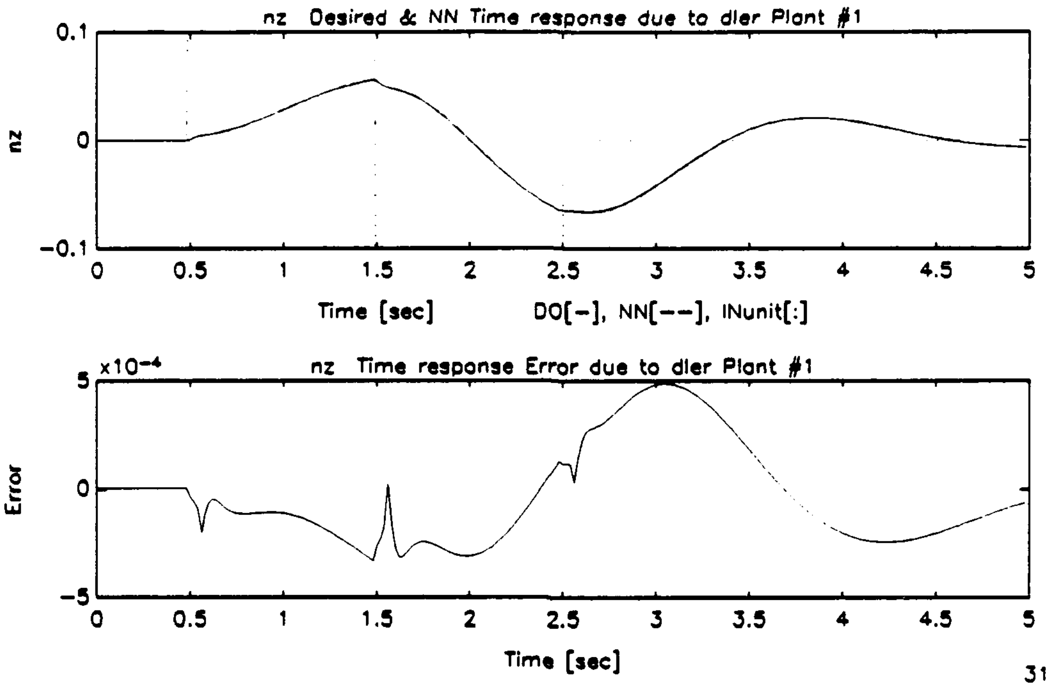


Figure B.7: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



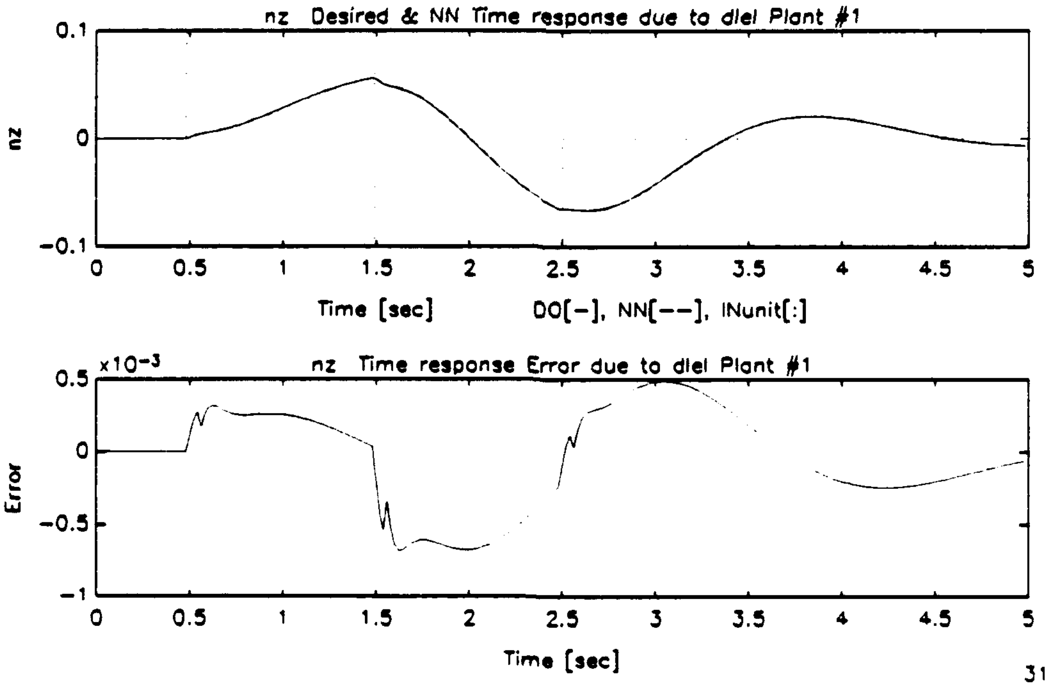
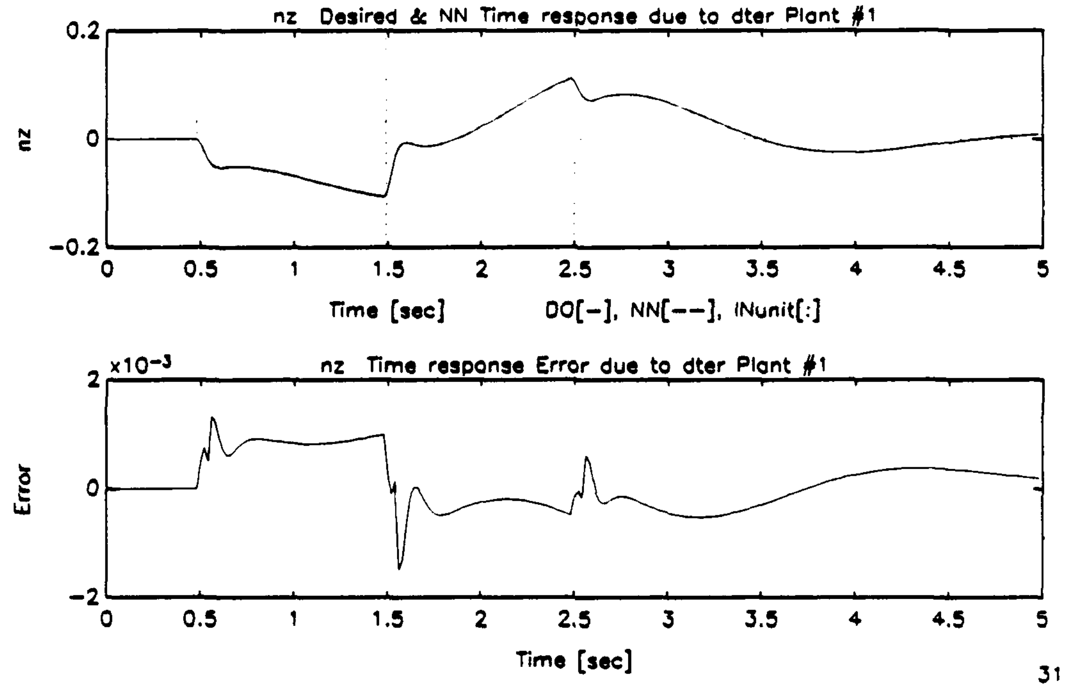


Figure B.8: **Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



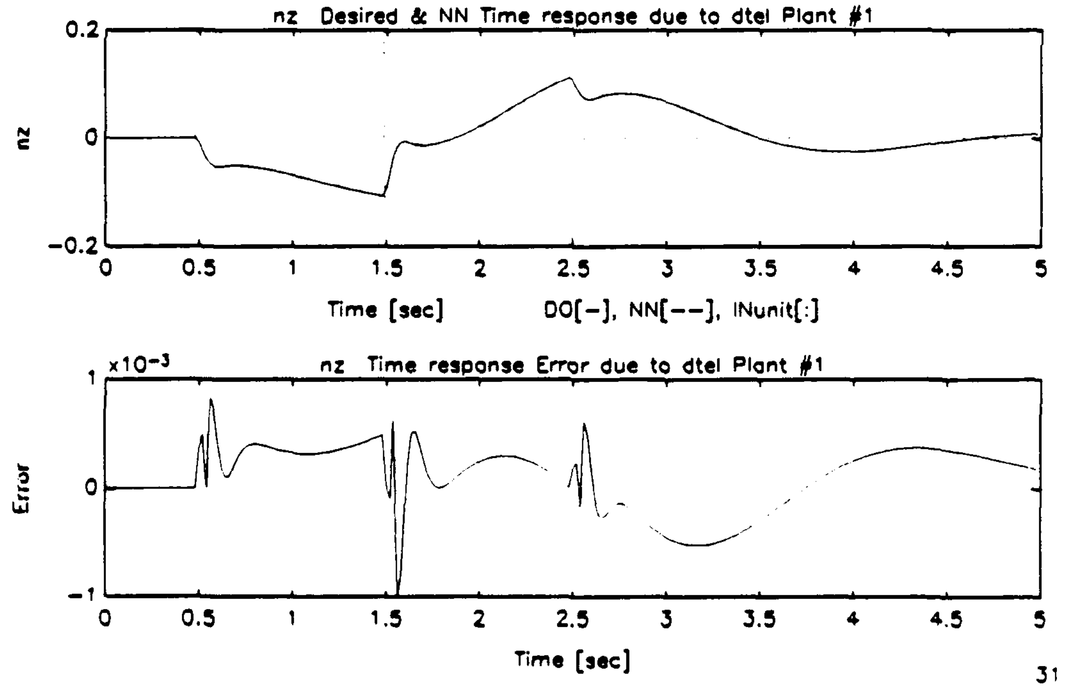
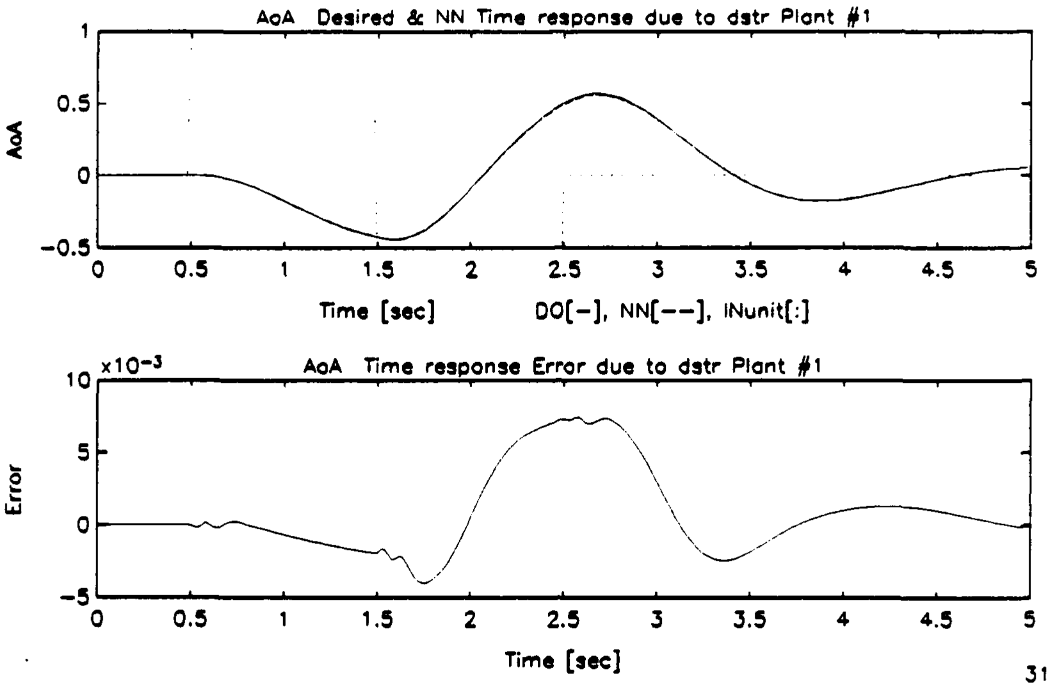


Figure B.9: **Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



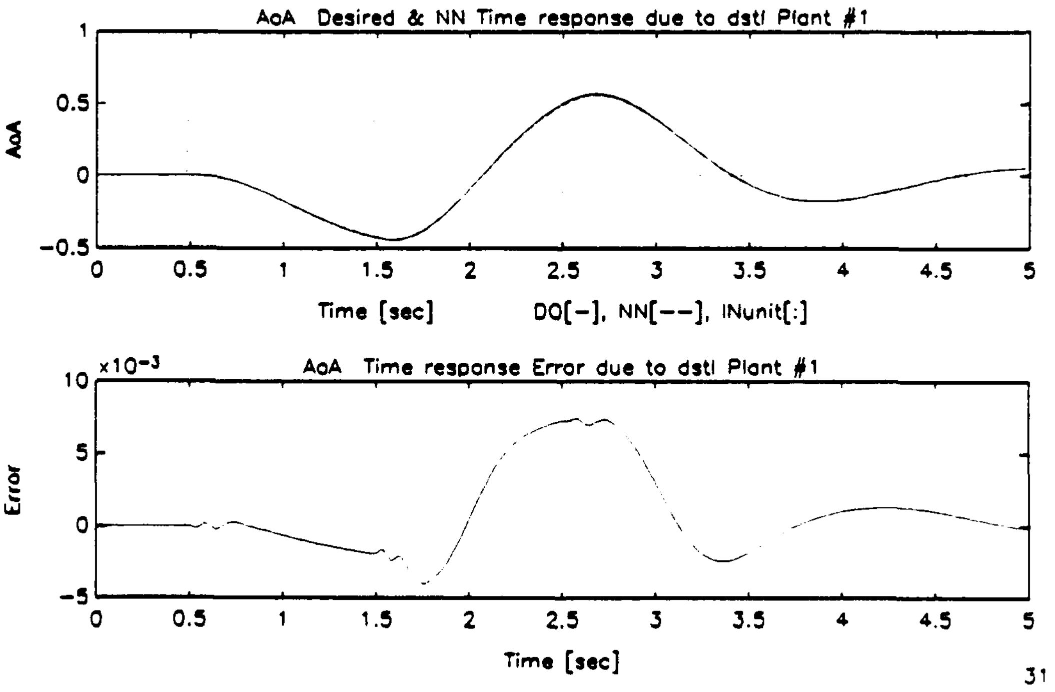


Figure B.10: Angle of Attack Response of Undamaged Aircraft Due to Stabilators, Dipole Command, Single Plant, Single Network Emulation

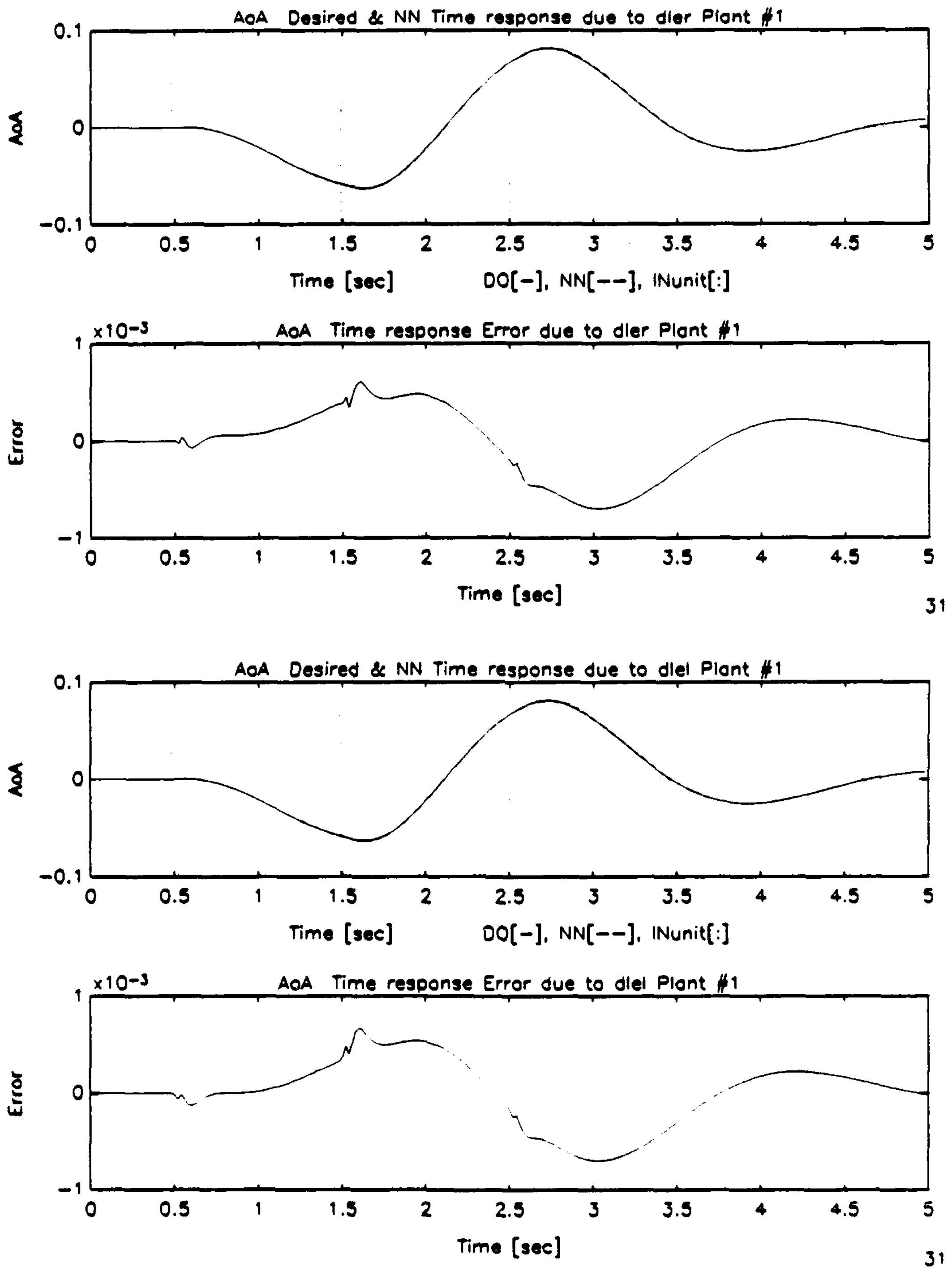
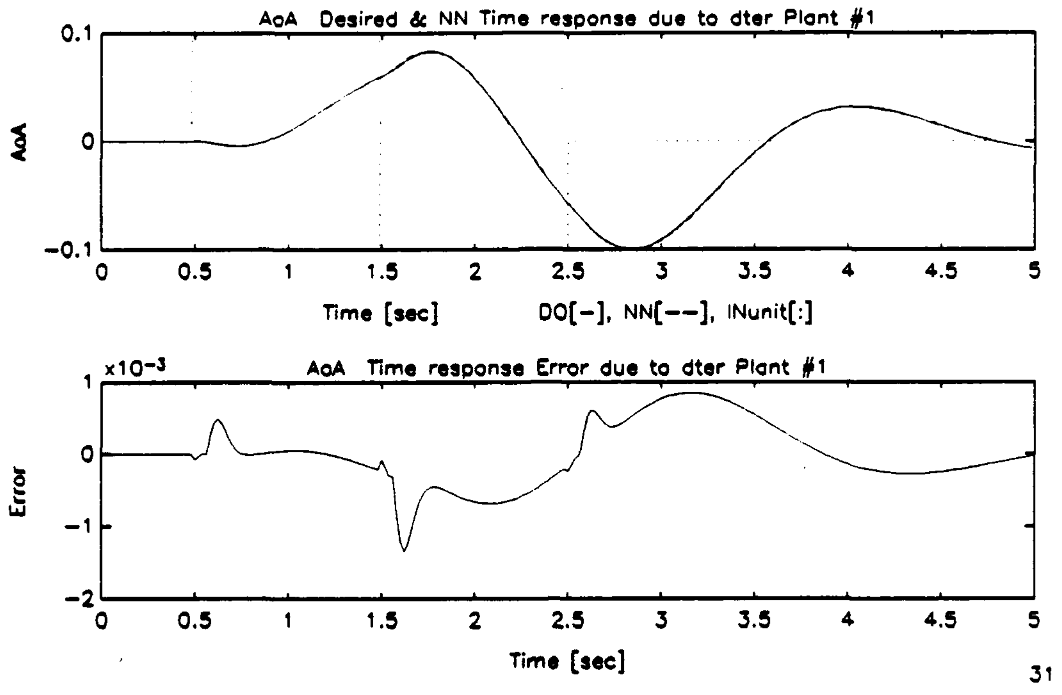


Figure B.11: Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation



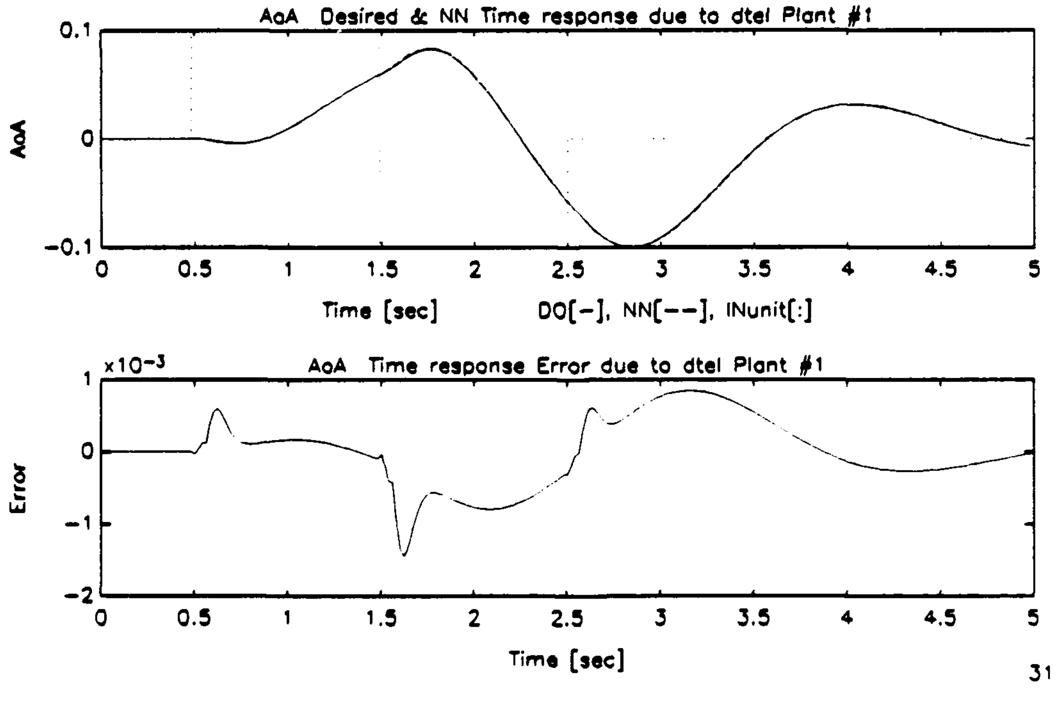
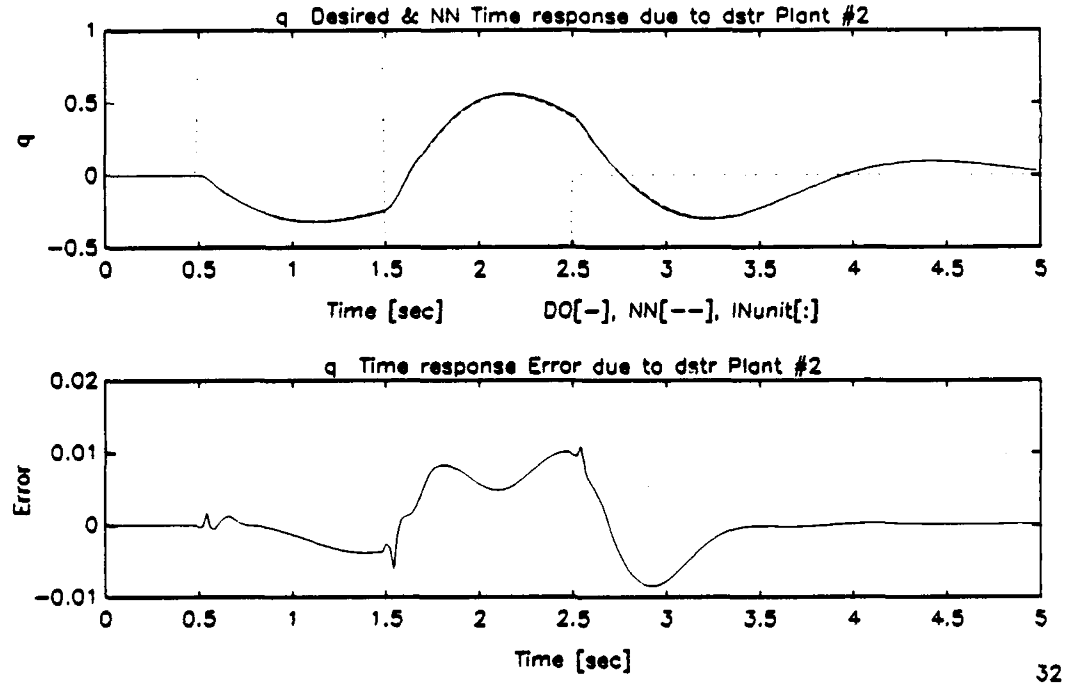


Figure B.12: Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation



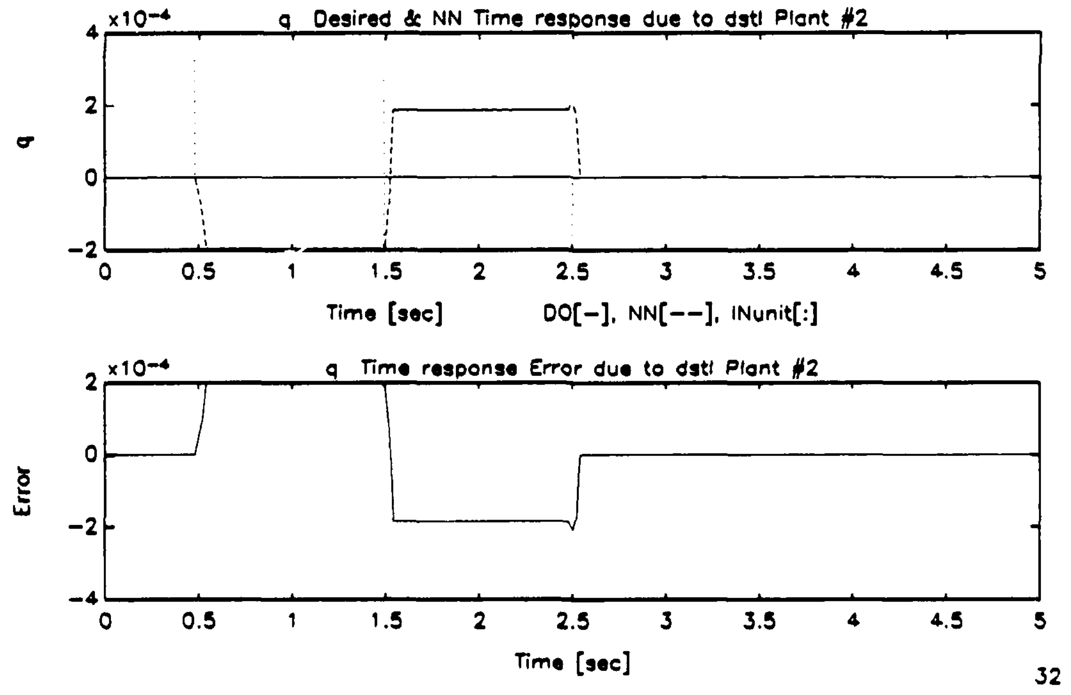


Figure B.13: **Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation**

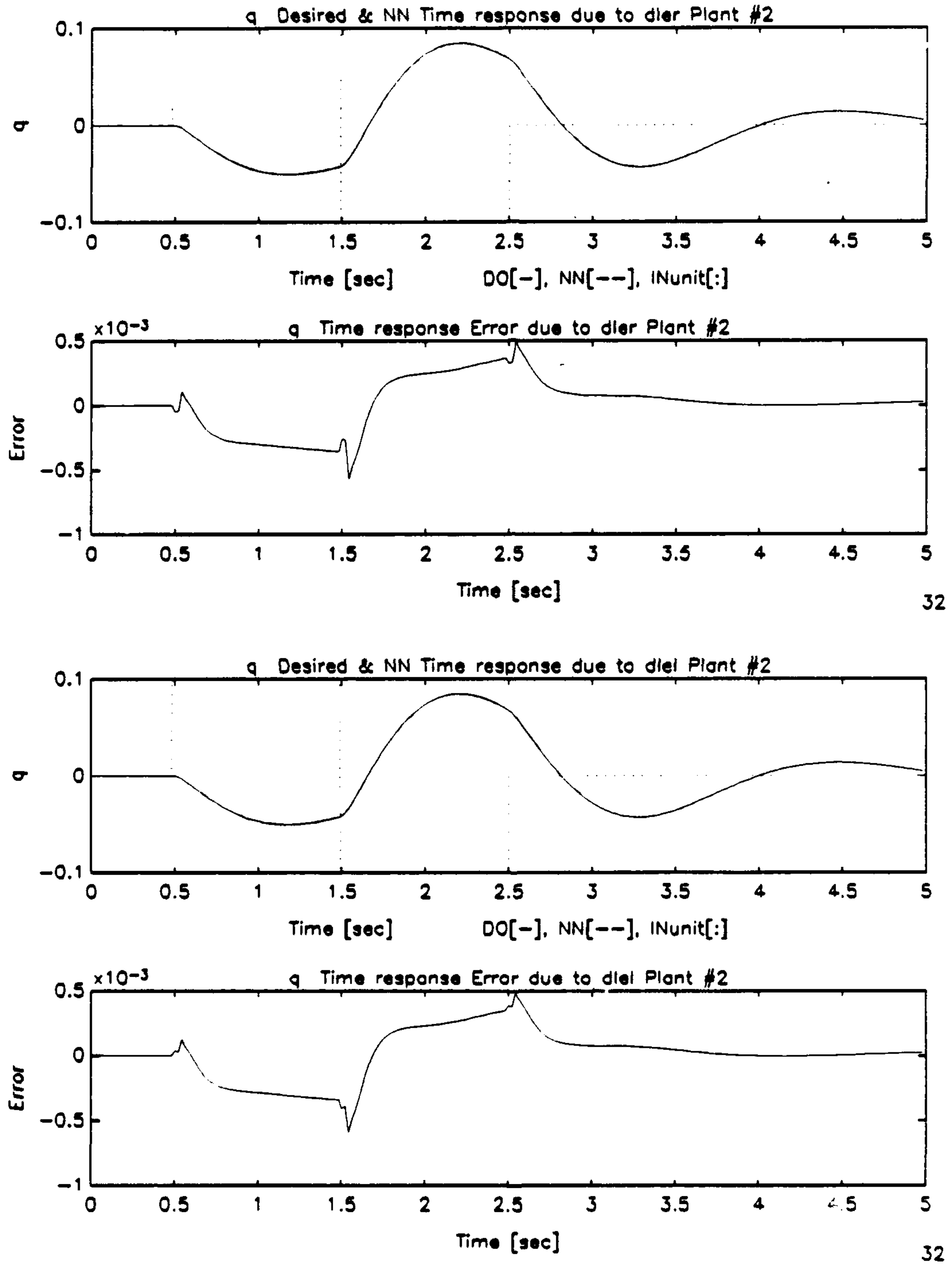


Figure B.14: **Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation**

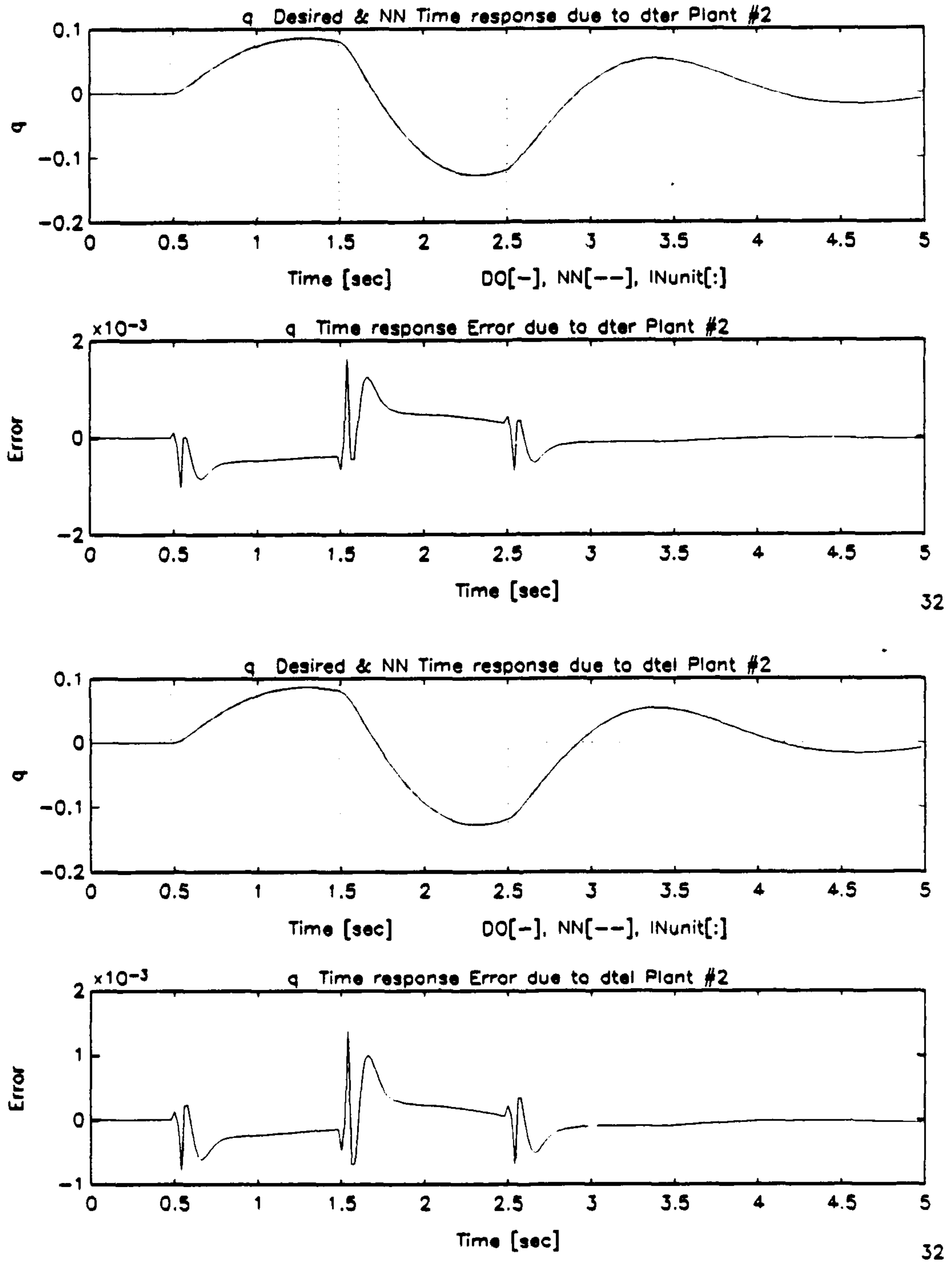
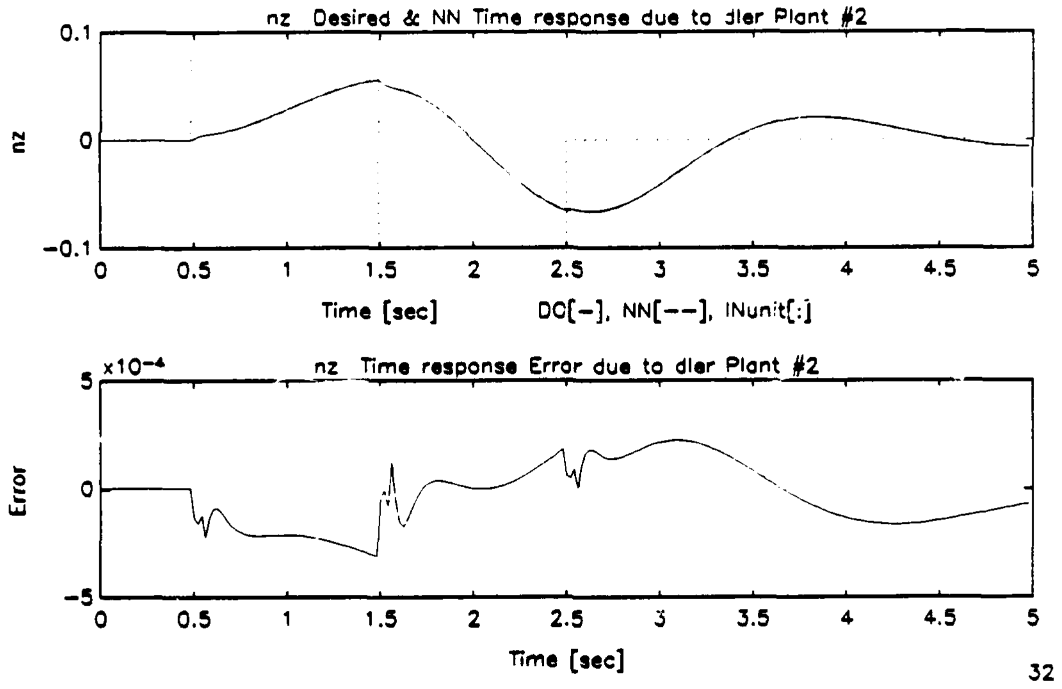


Figure B.15: **Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



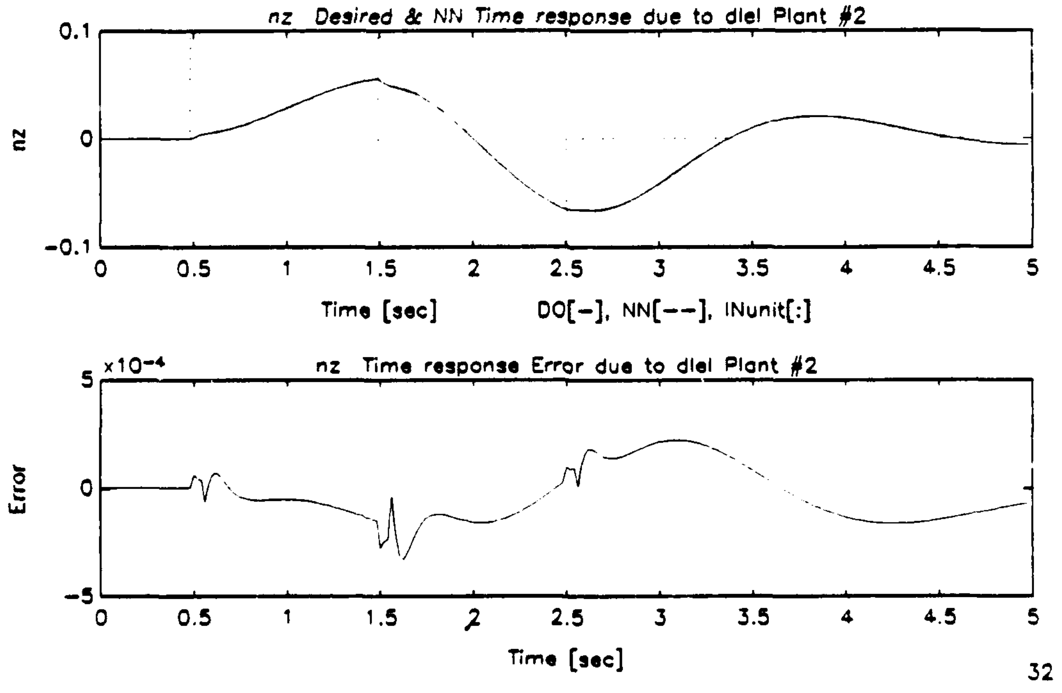


Figure B.16: Normal Acceleration Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation

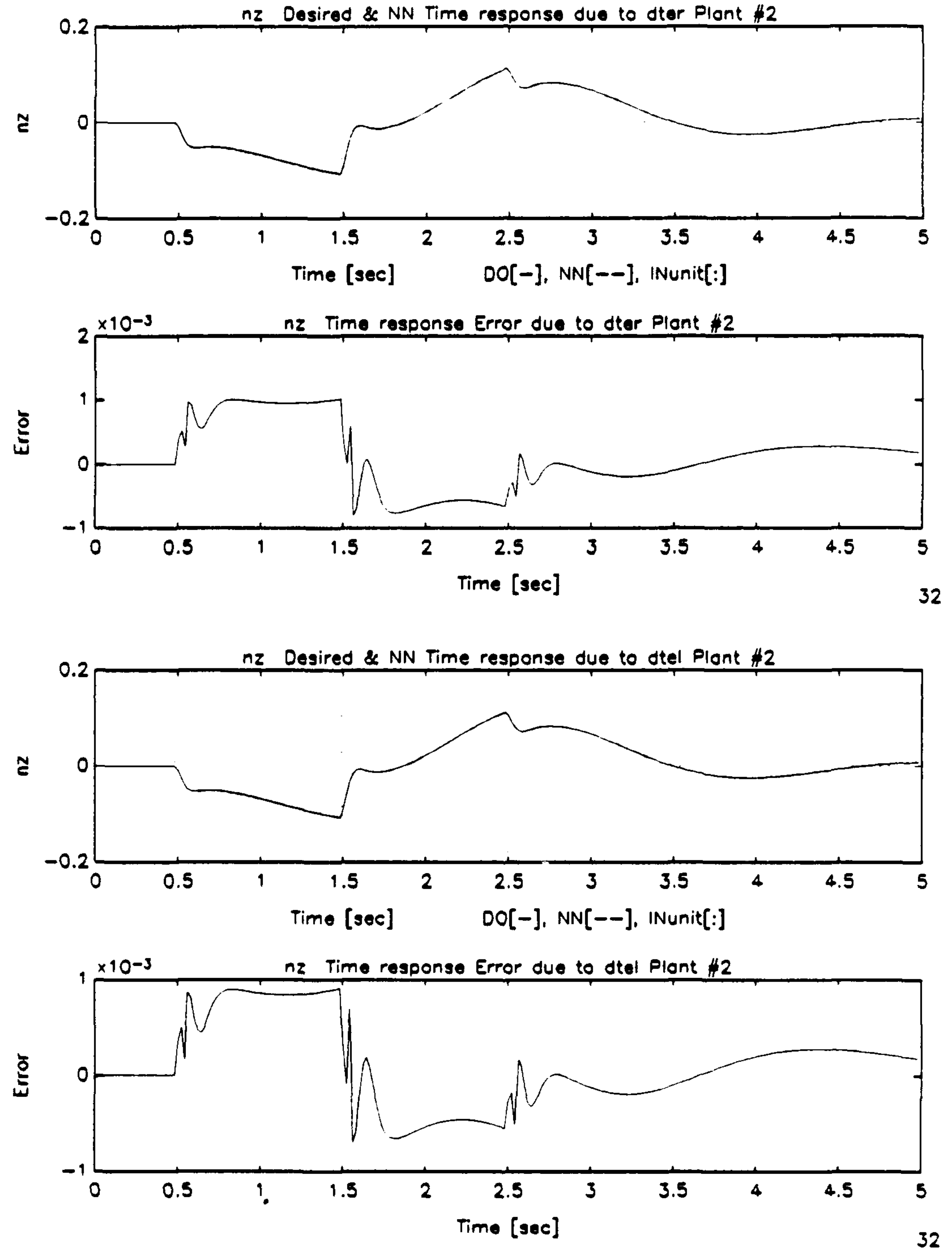
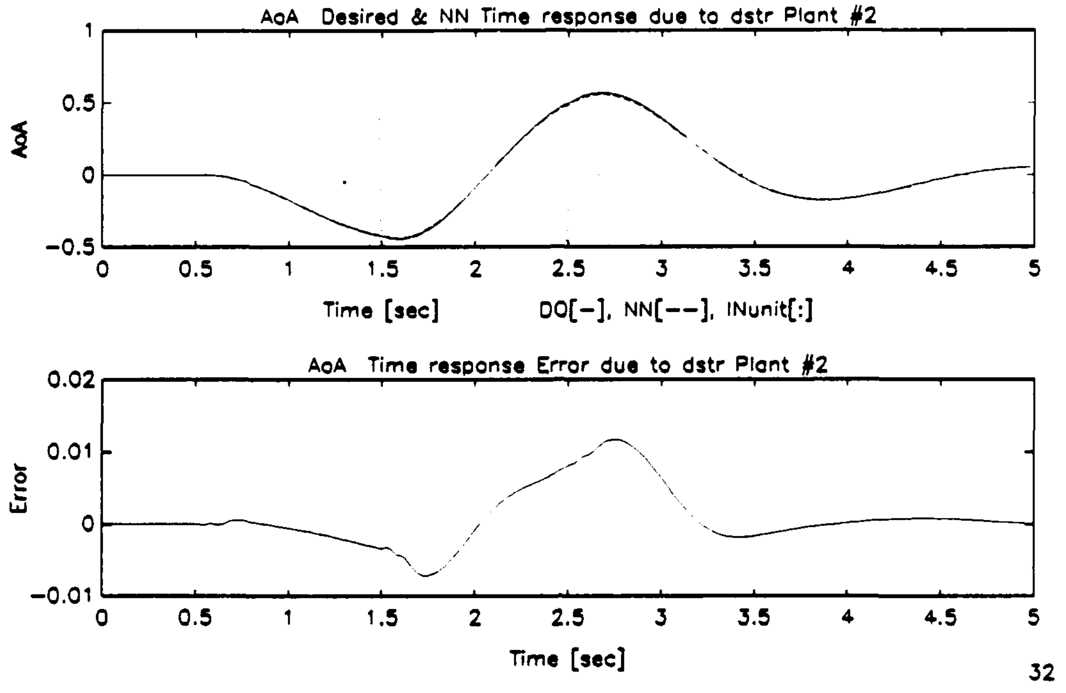


Figure B.17: **Normal Acceleration Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



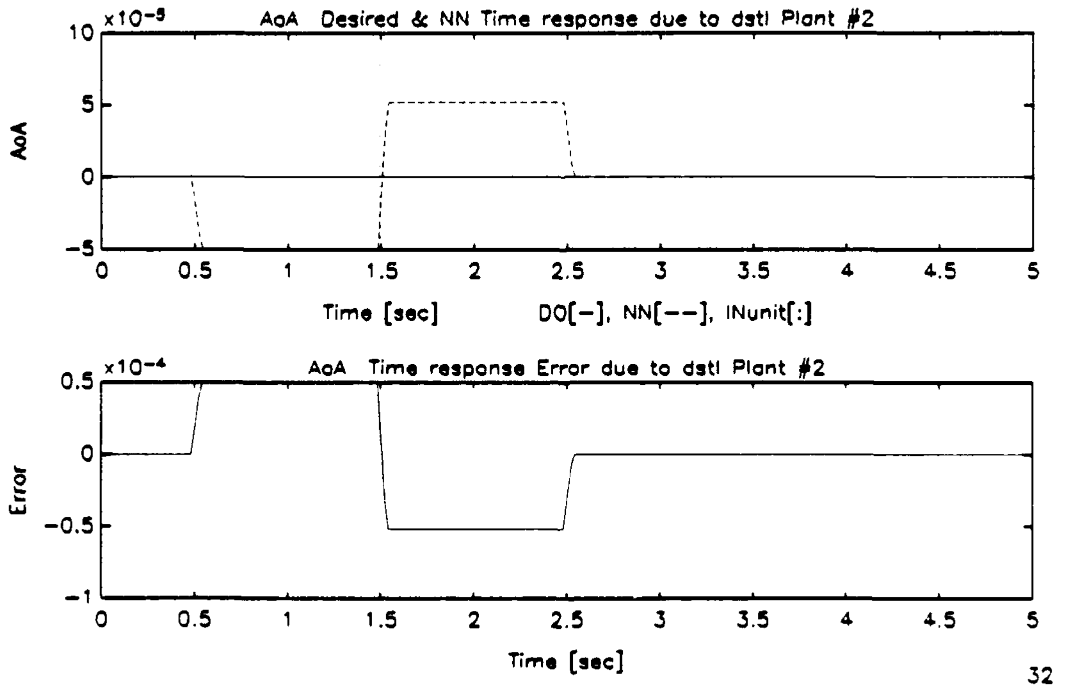


Figure B.18: **Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Single Plant, Single Network Emulation**

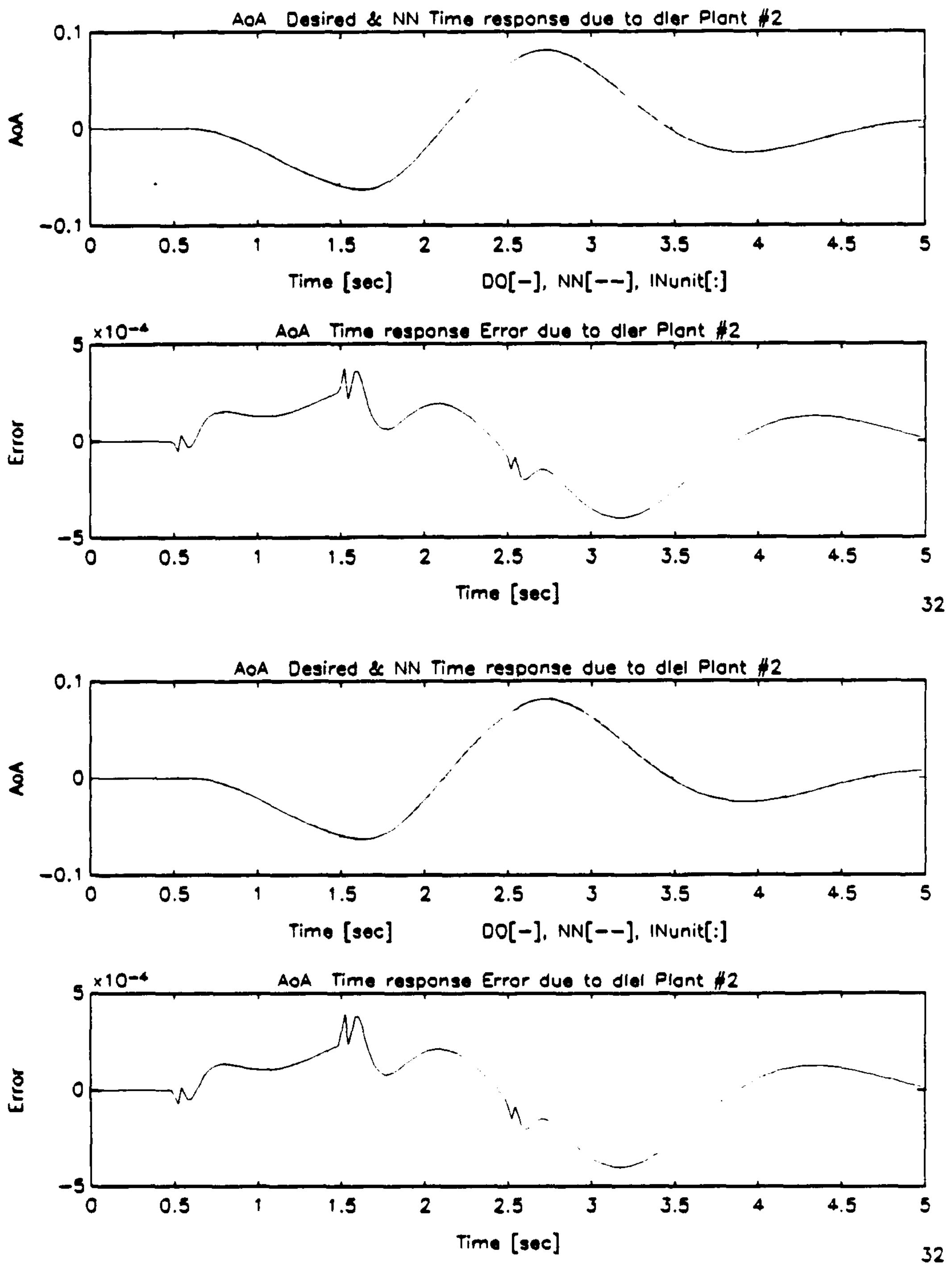
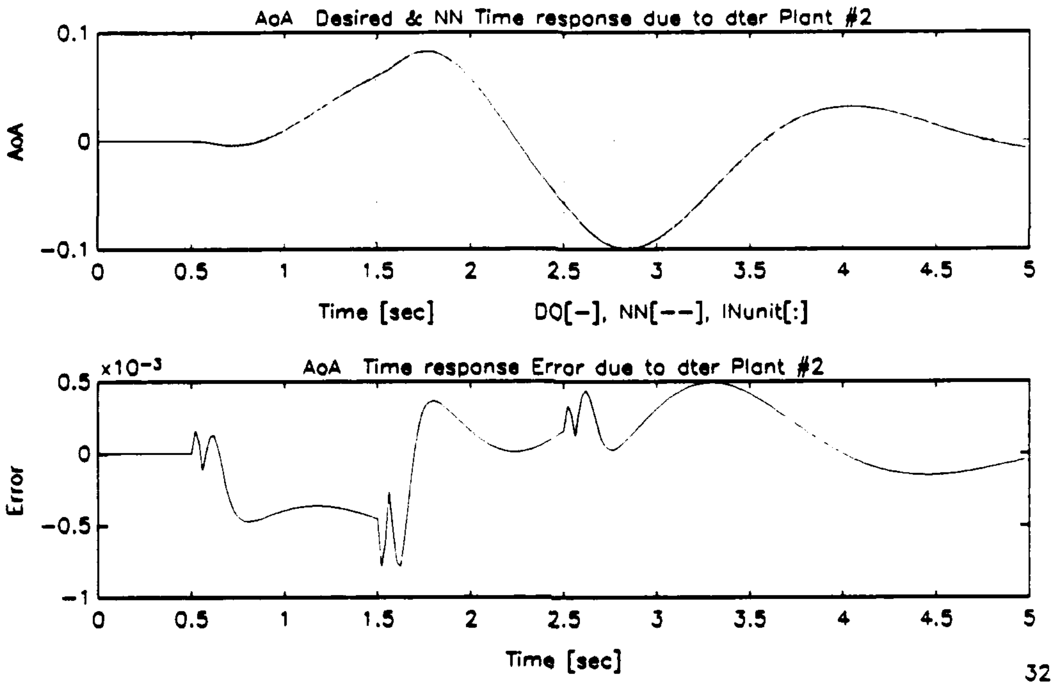


Figure B.19: **Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Single Plant, Single Network Emulation**



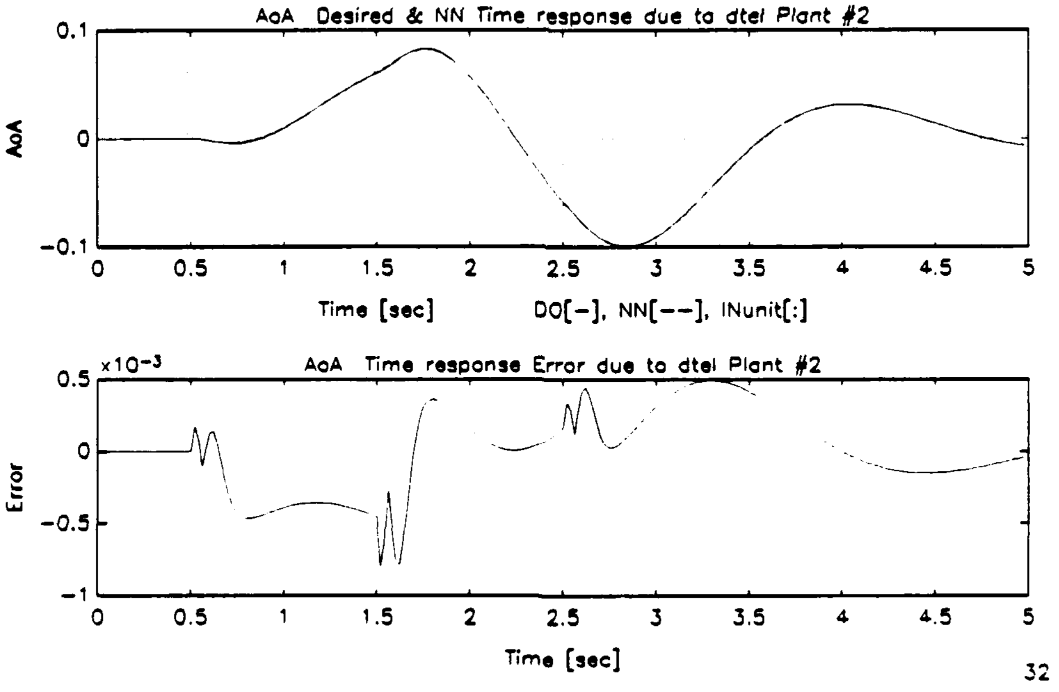


Figure B.20: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Single Plant, Single Network Emulation**

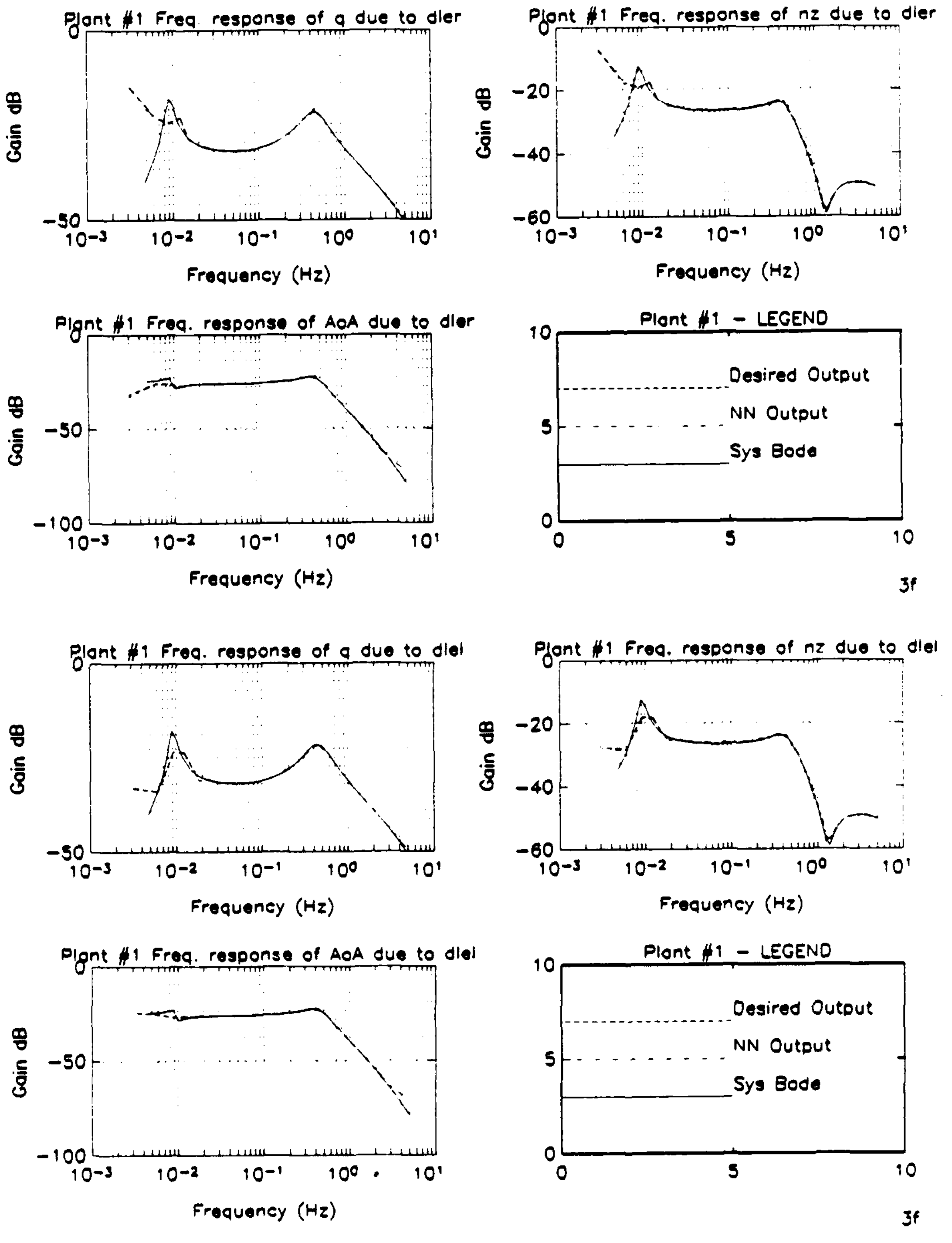


Figure B.21: Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Single Network Emulation

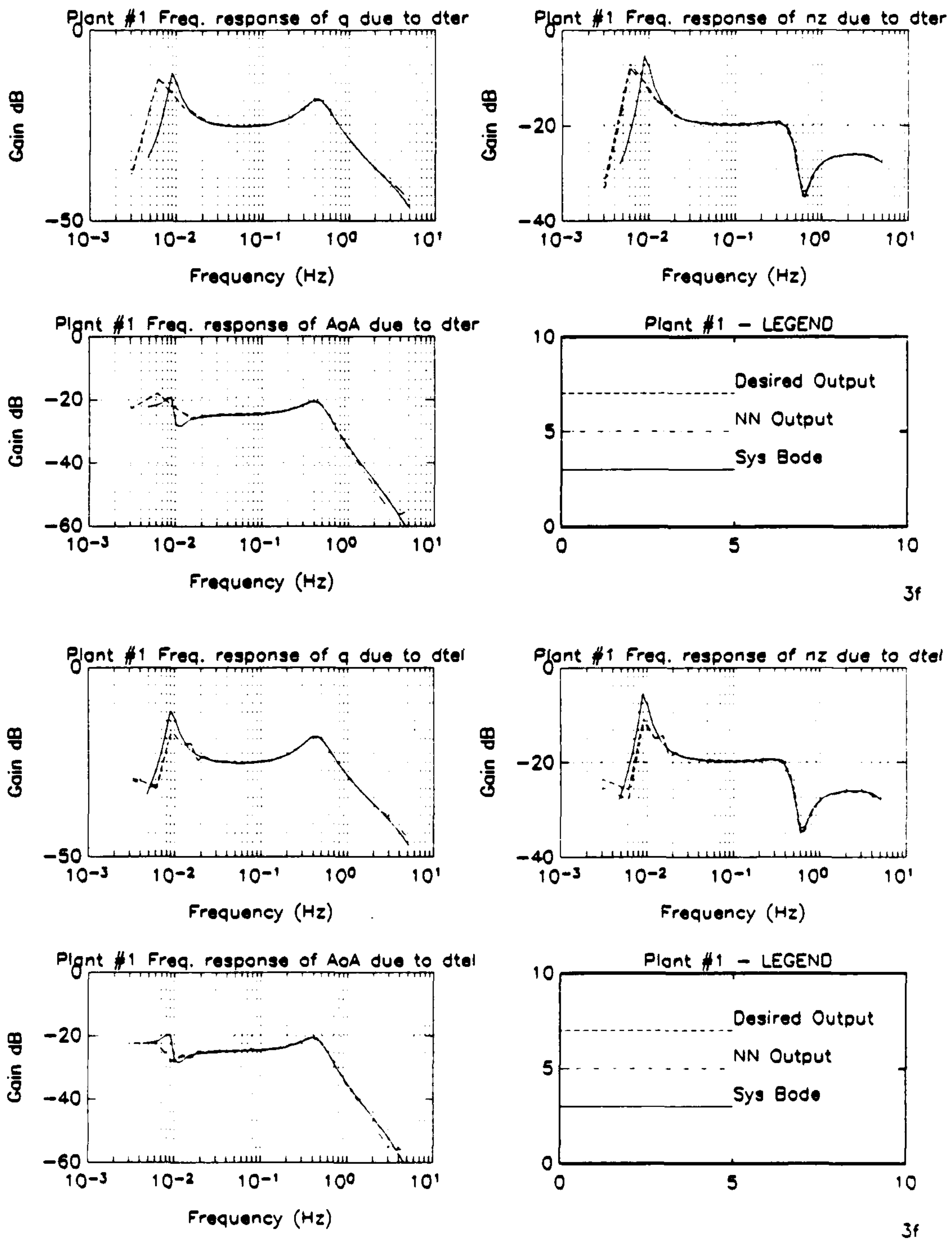


Figure B.22: Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Single Network Emulation

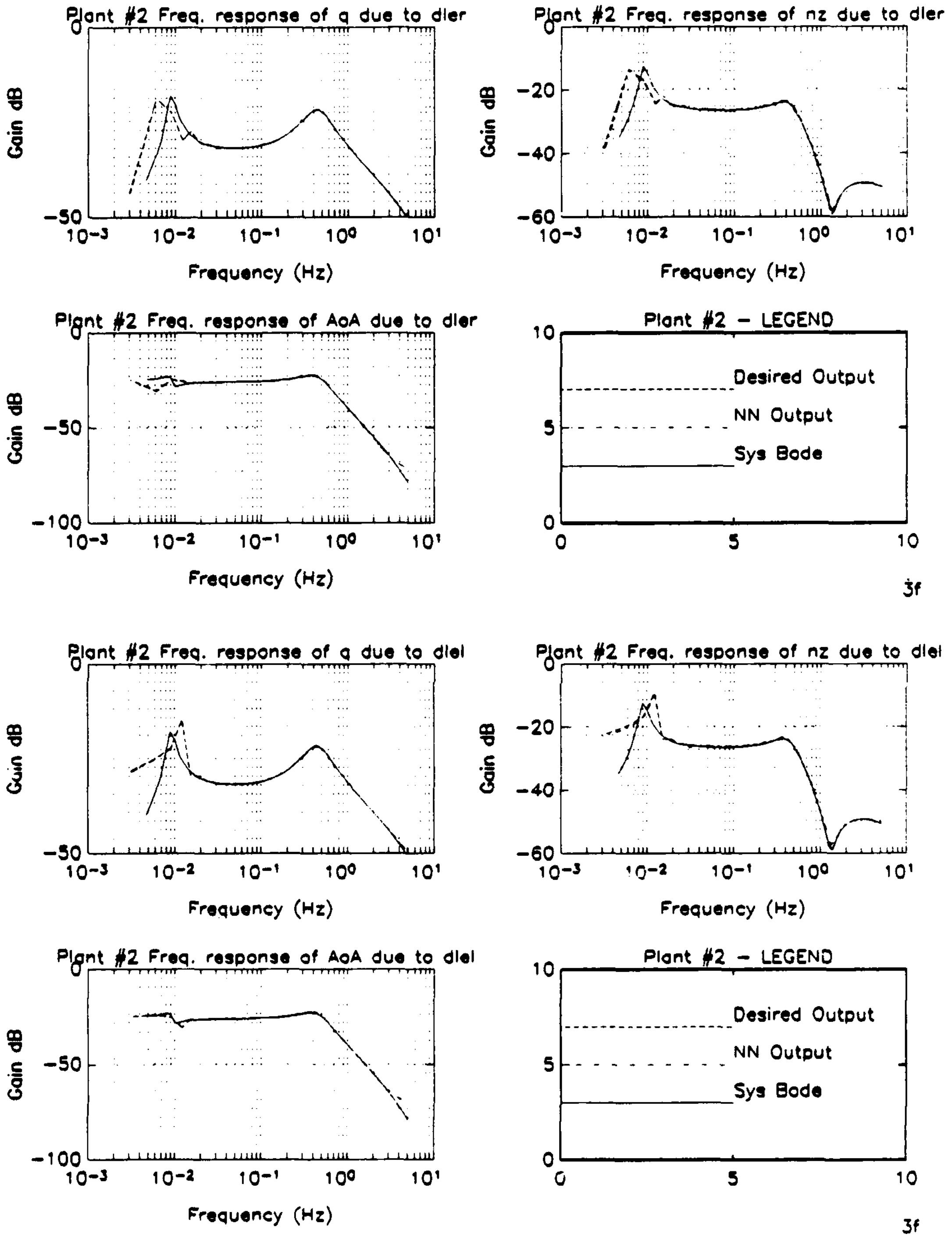


Figure B.23: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Single Network Emulation**

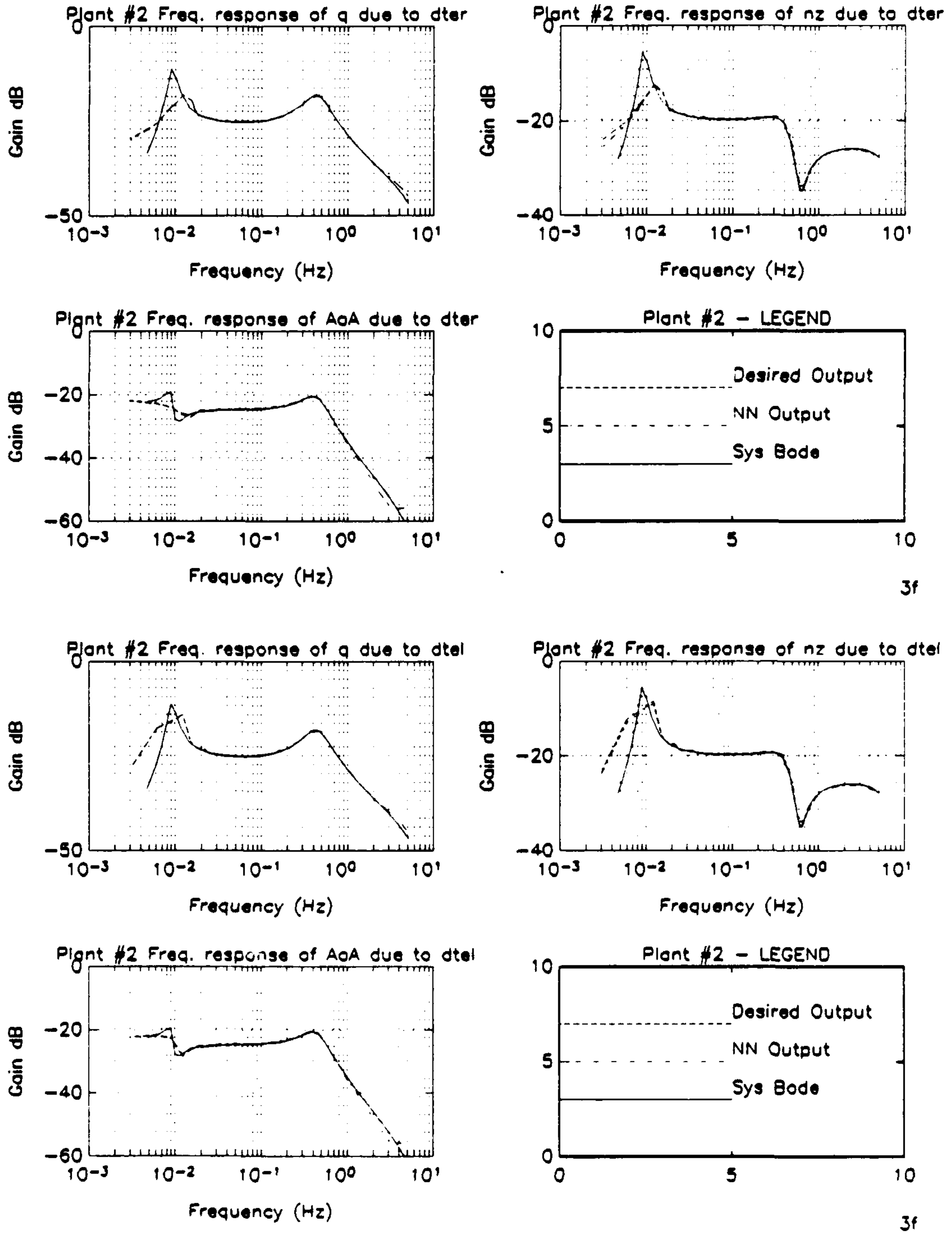


Figure B.24: Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Single Network Emulation

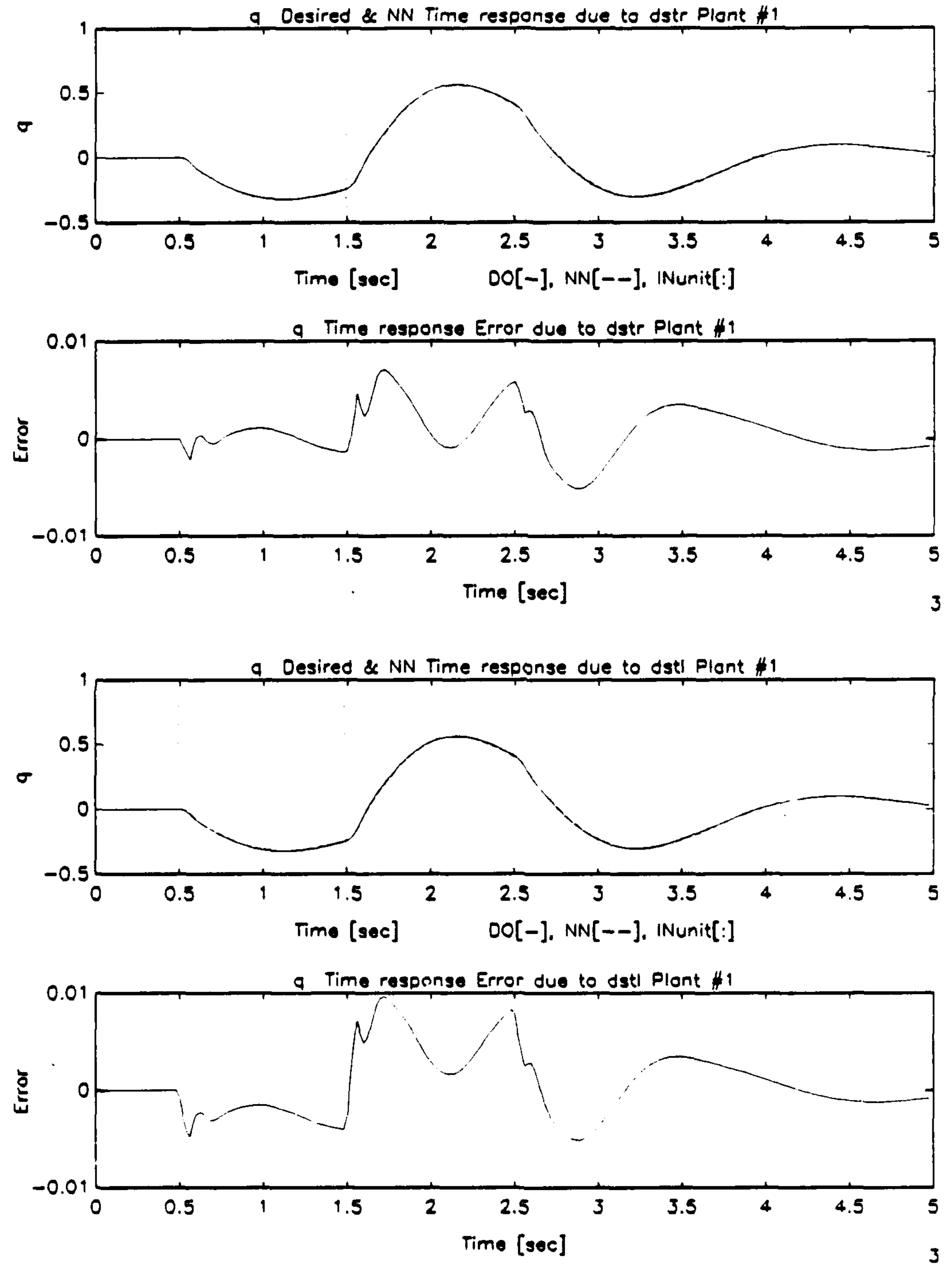


Figure B.25: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation**

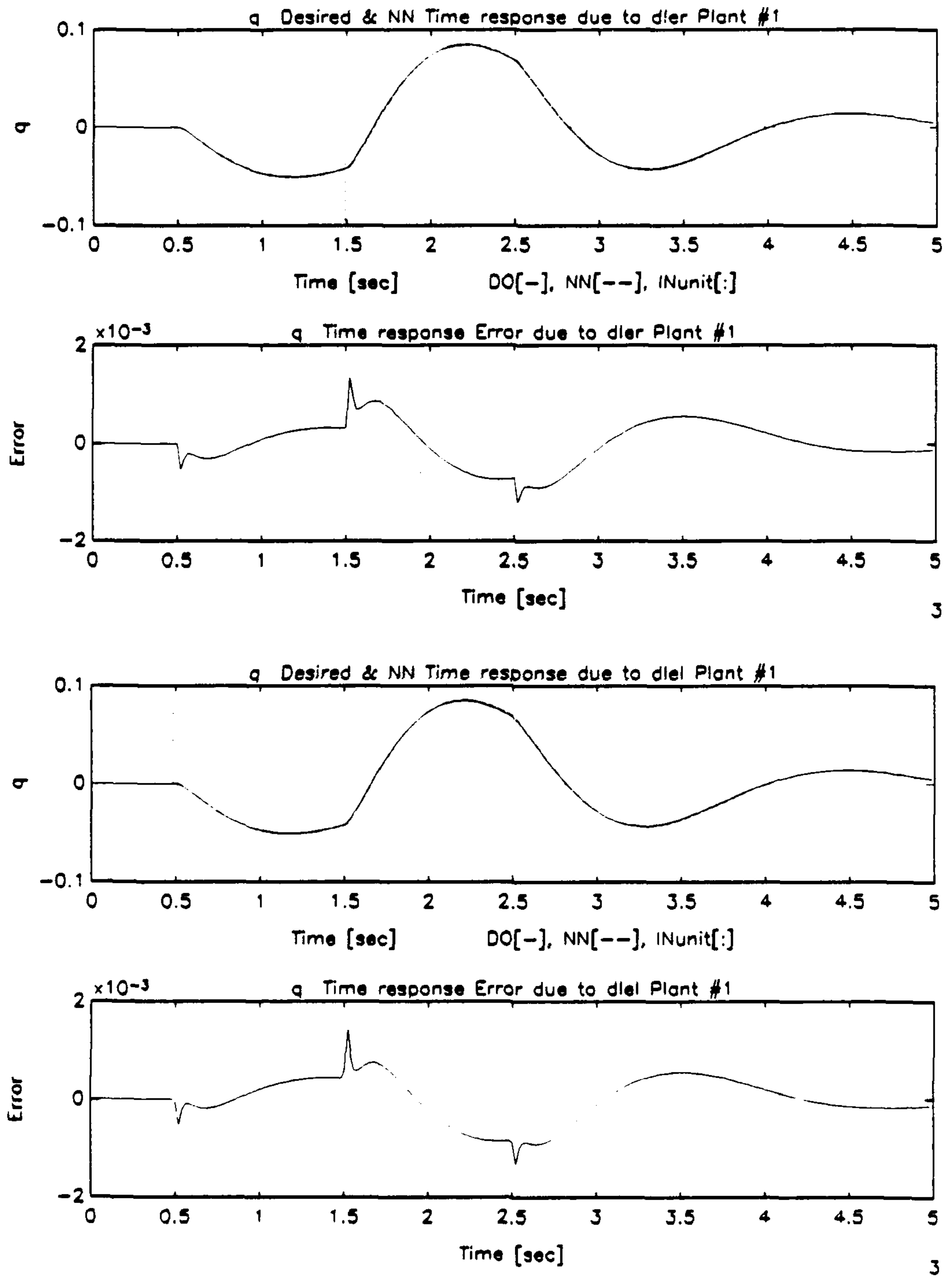
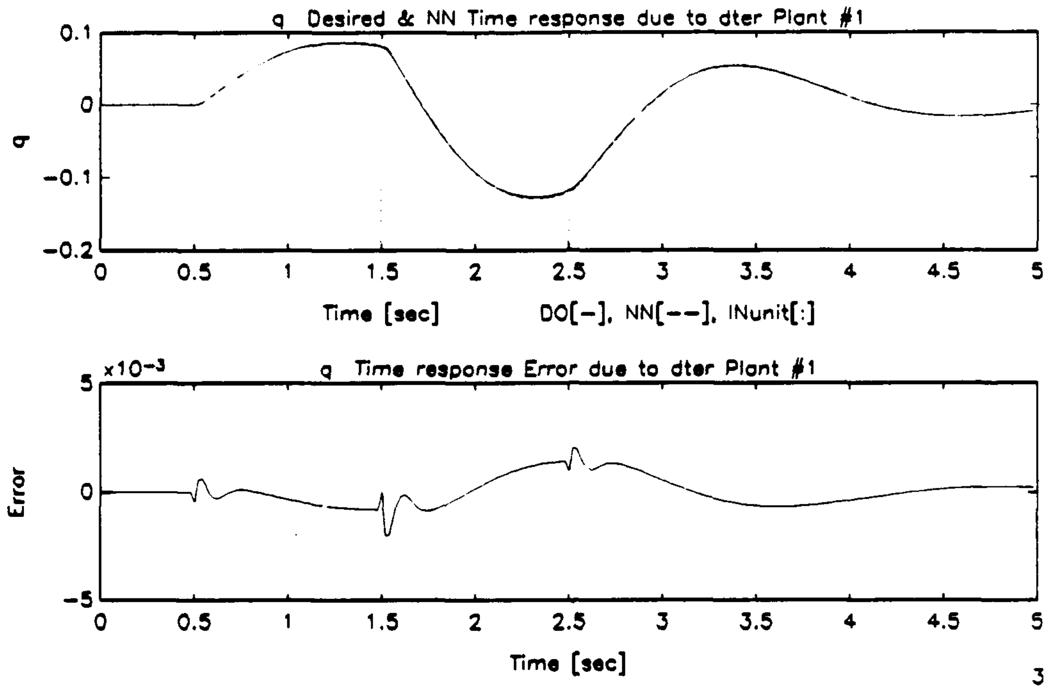


Figure B.26: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



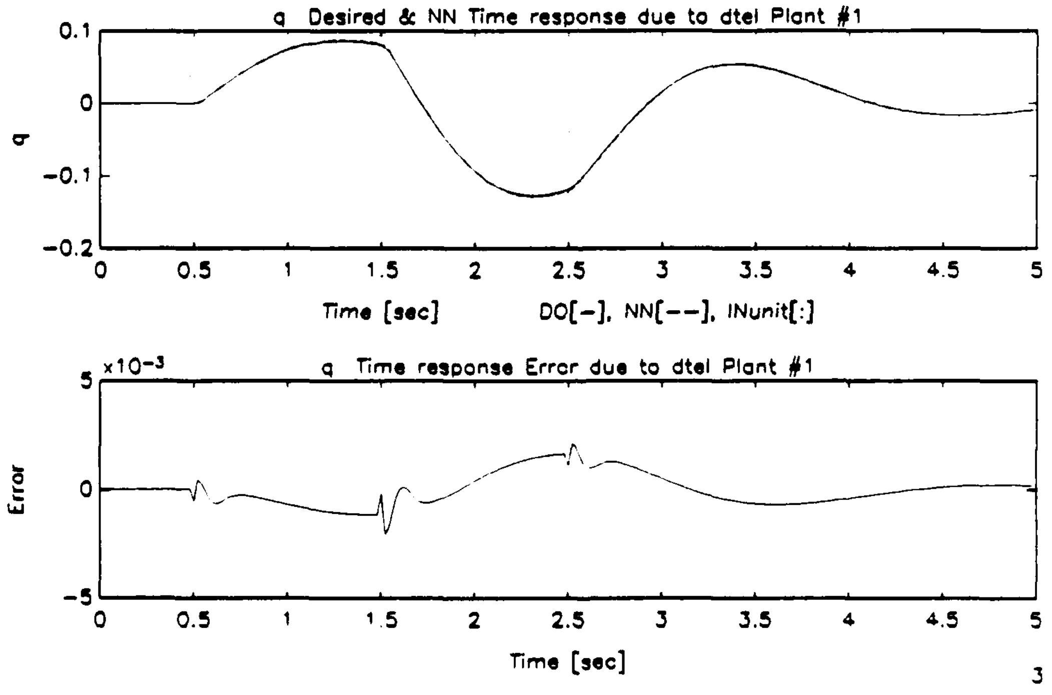


Figure B.27: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**

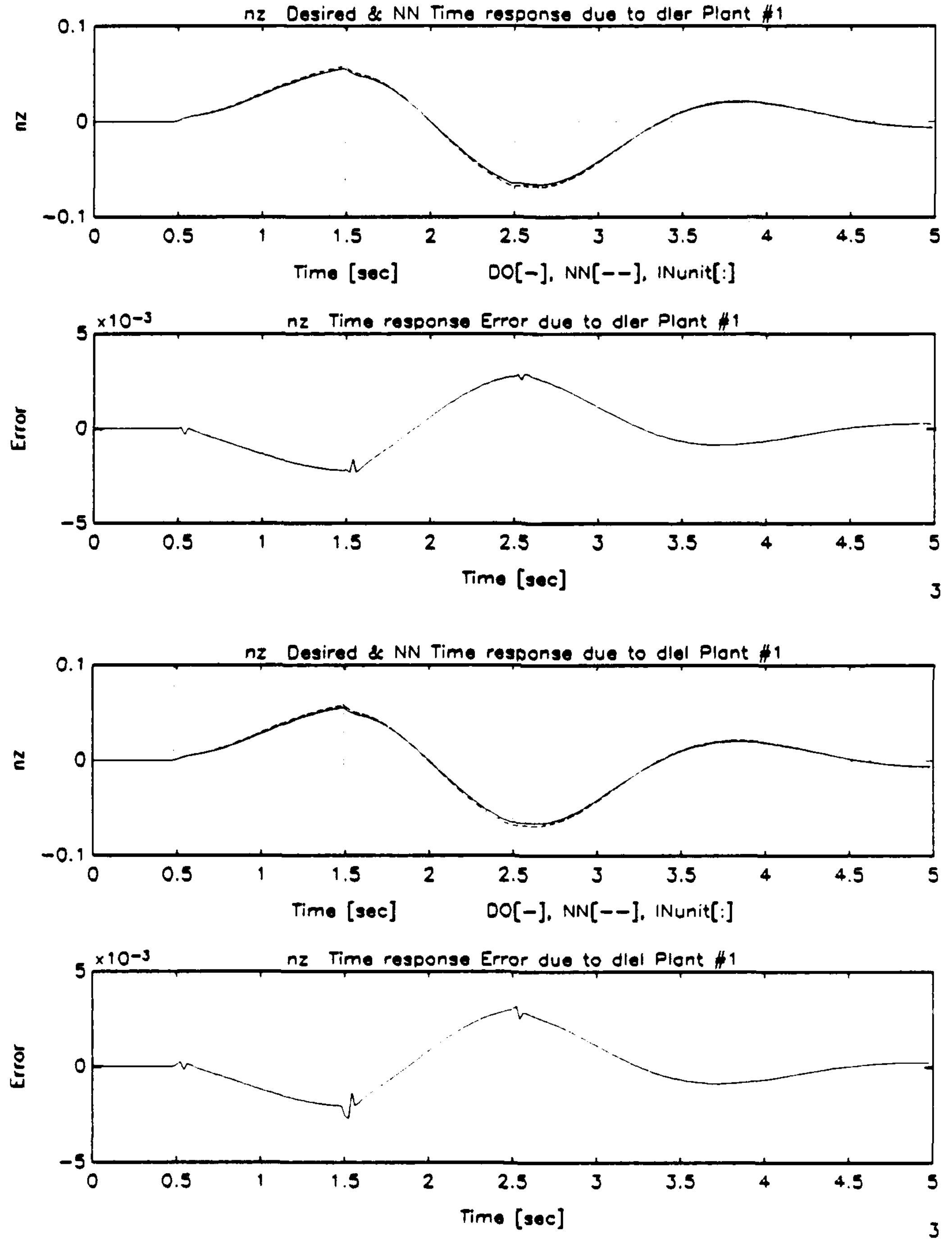
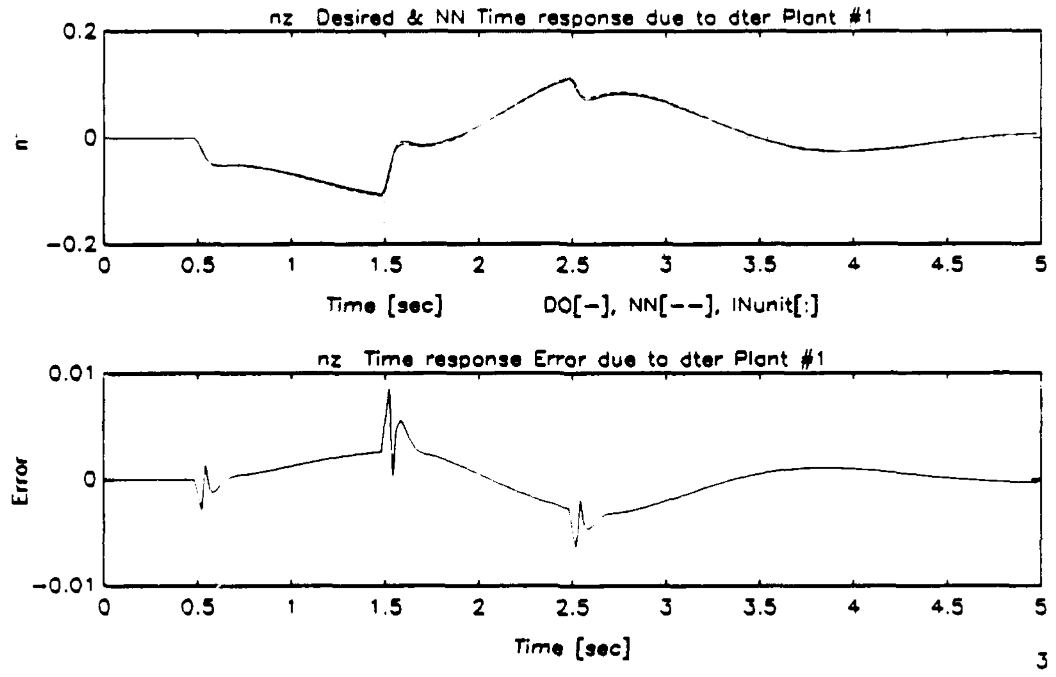


Figure B.28: **Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



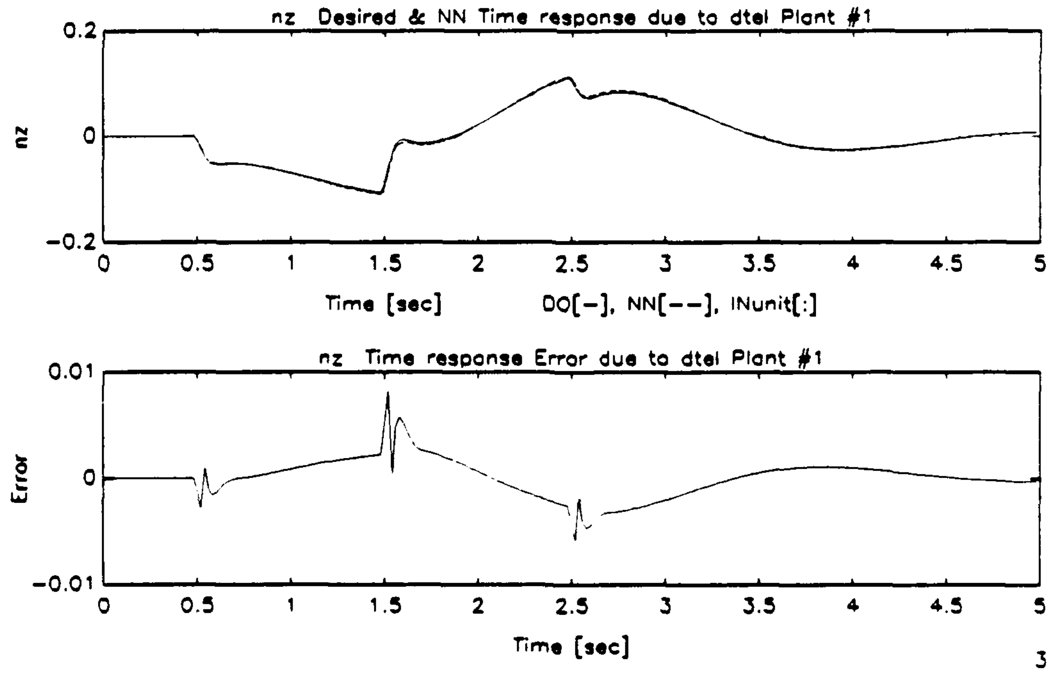


Figure B.29: **Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**

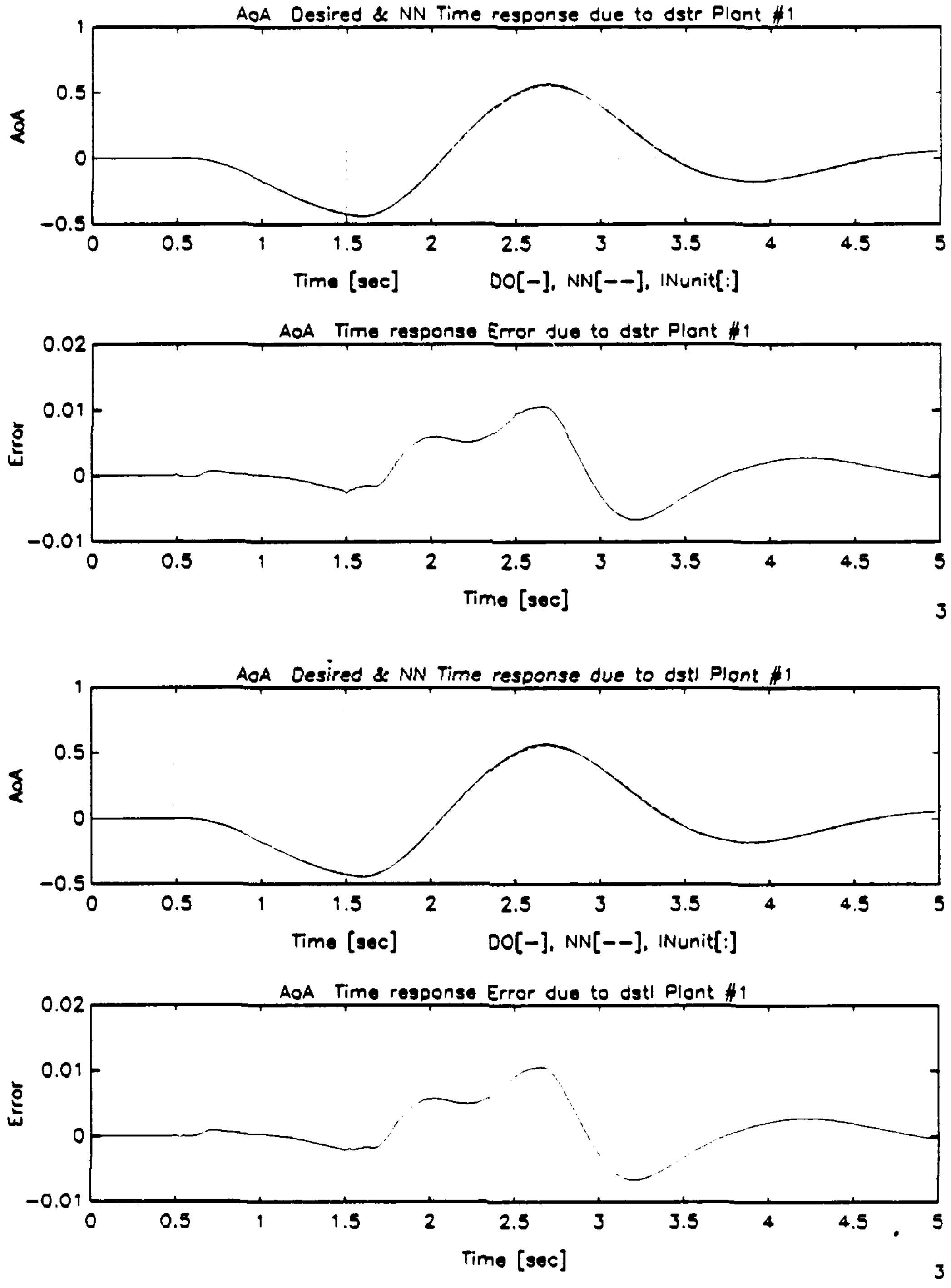
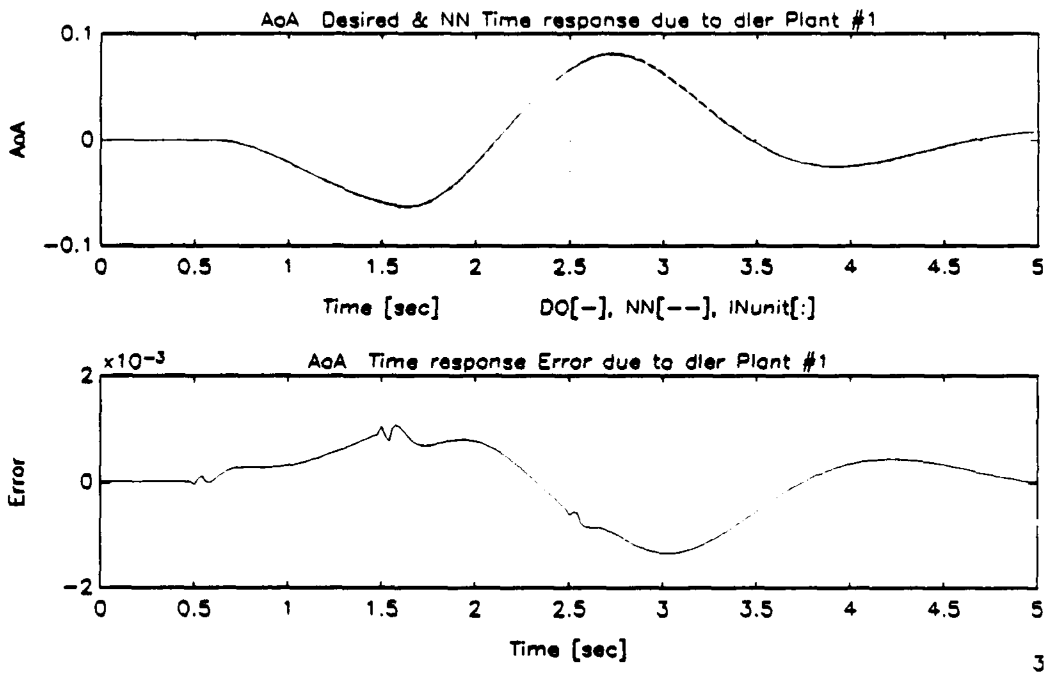


Figure B.30: **Angle of Attack Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation**



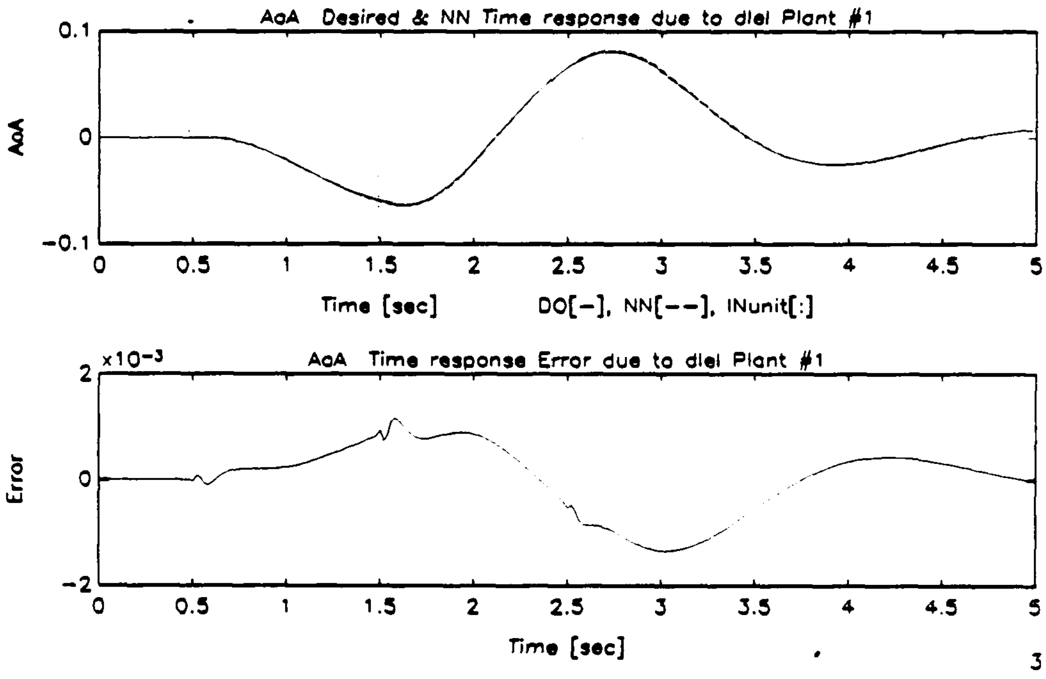
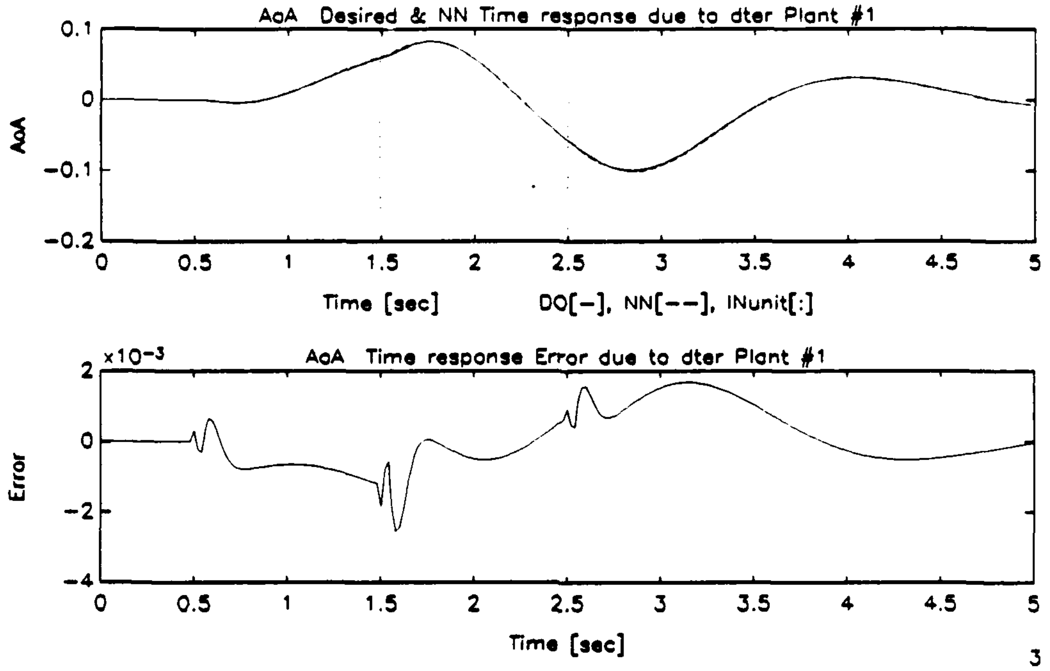


Figure B.31: **Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



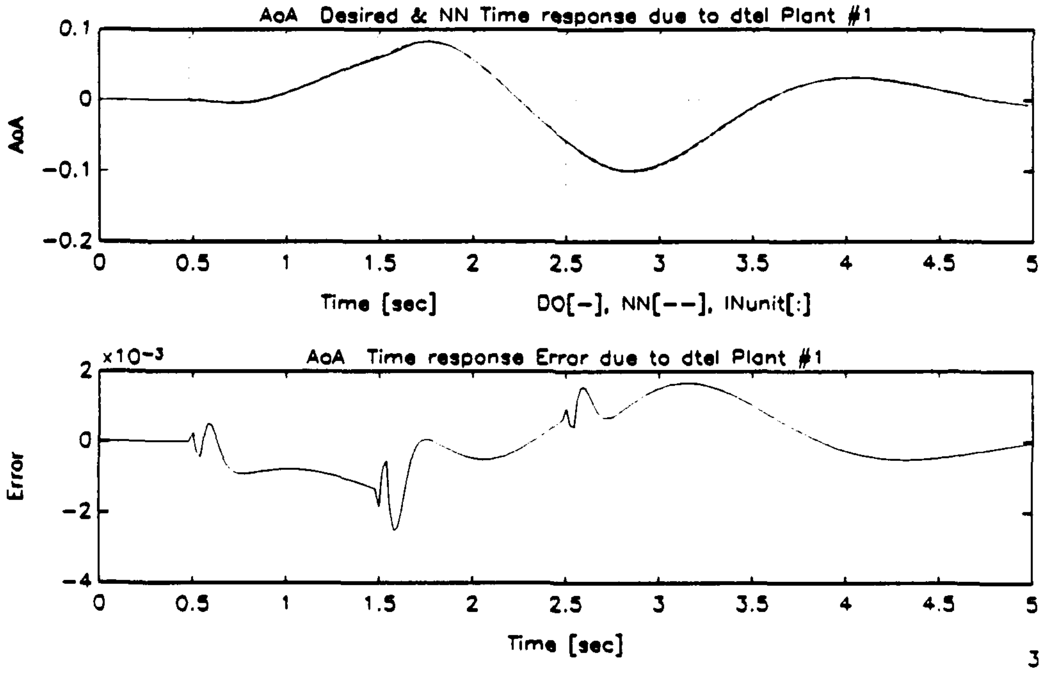


Figure B.32: **Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**

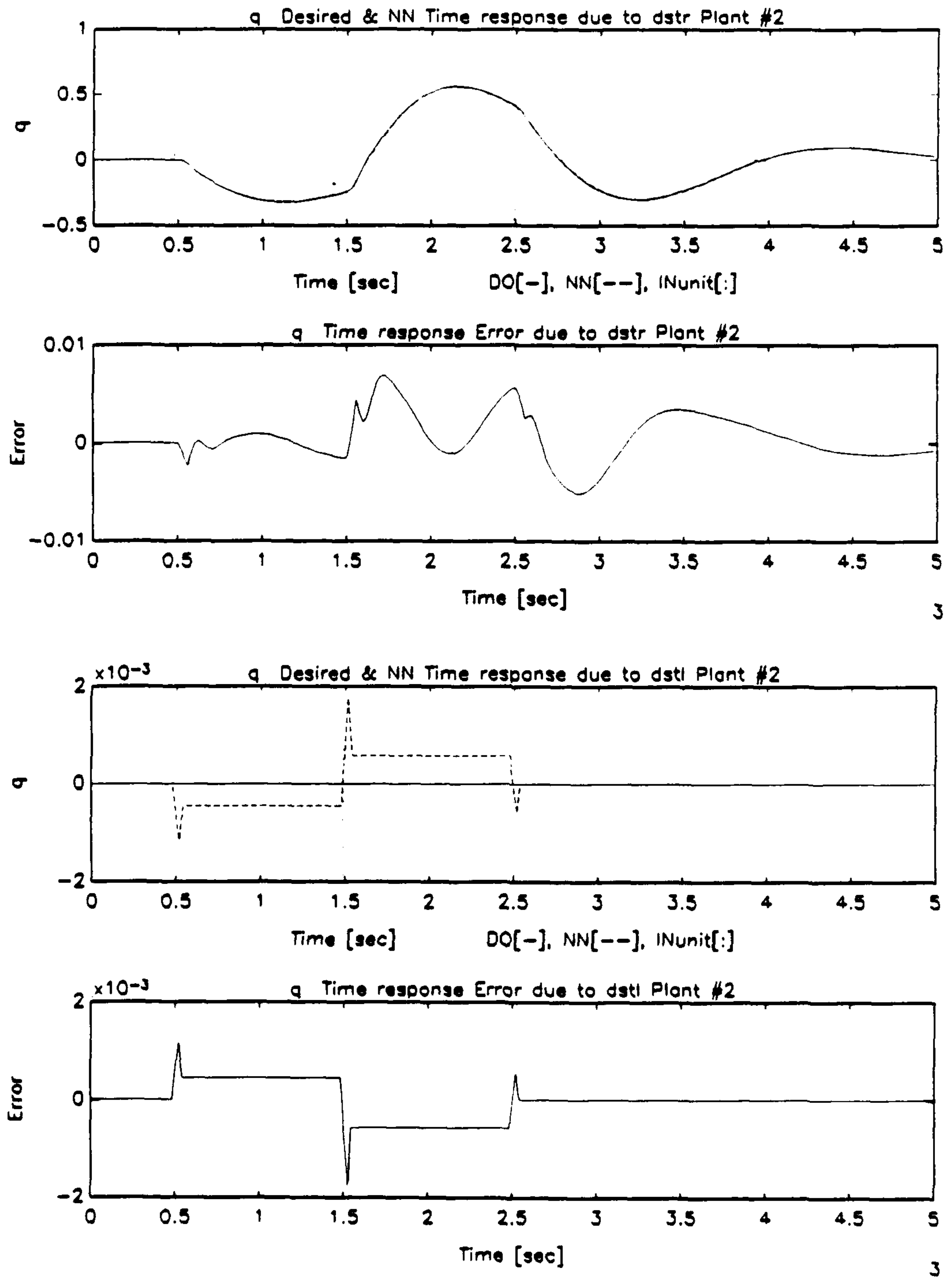


Figure B.33: Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation

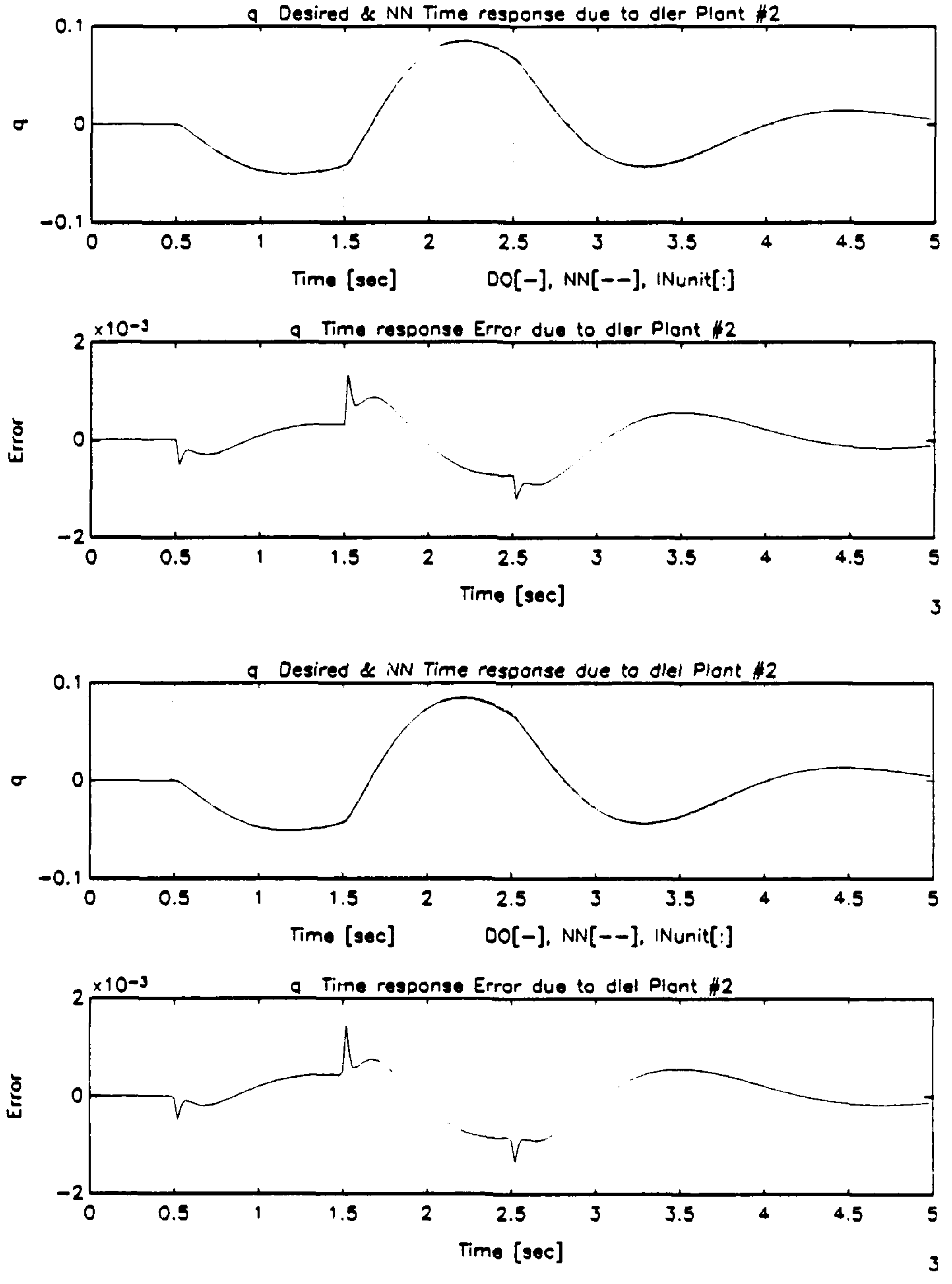
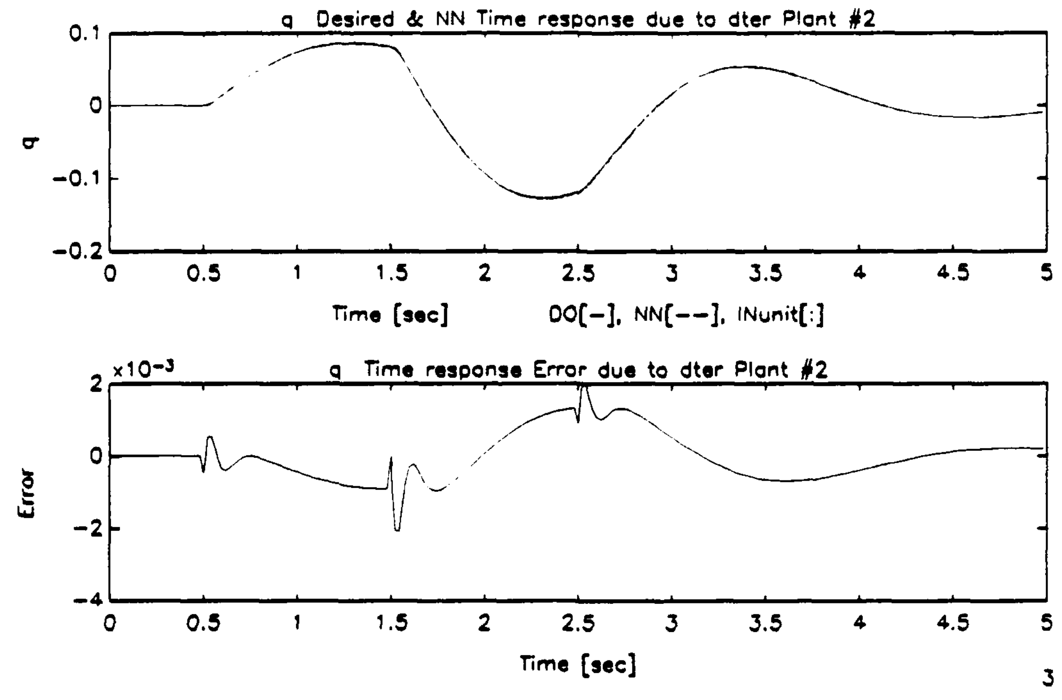


Figure B.34: **Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



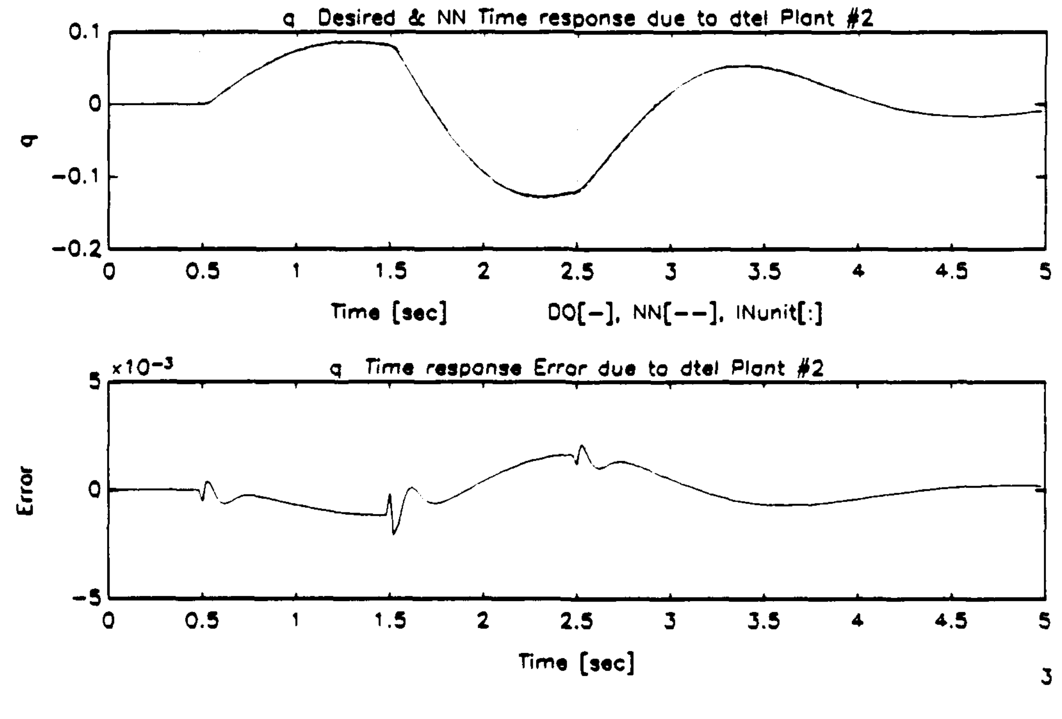
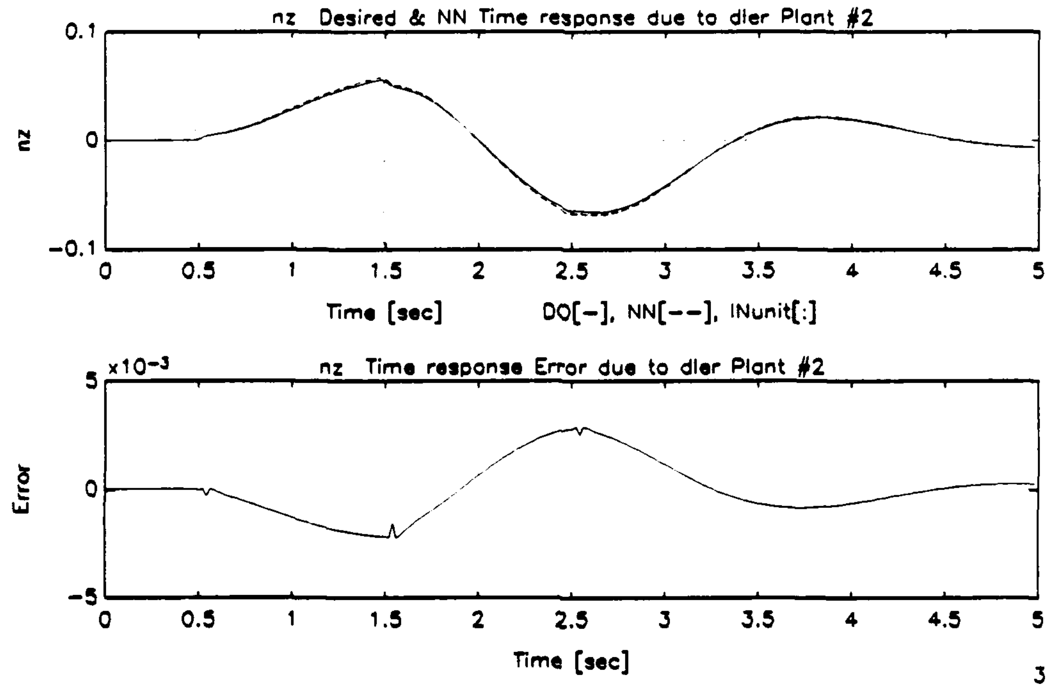


Figure B.35: **Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



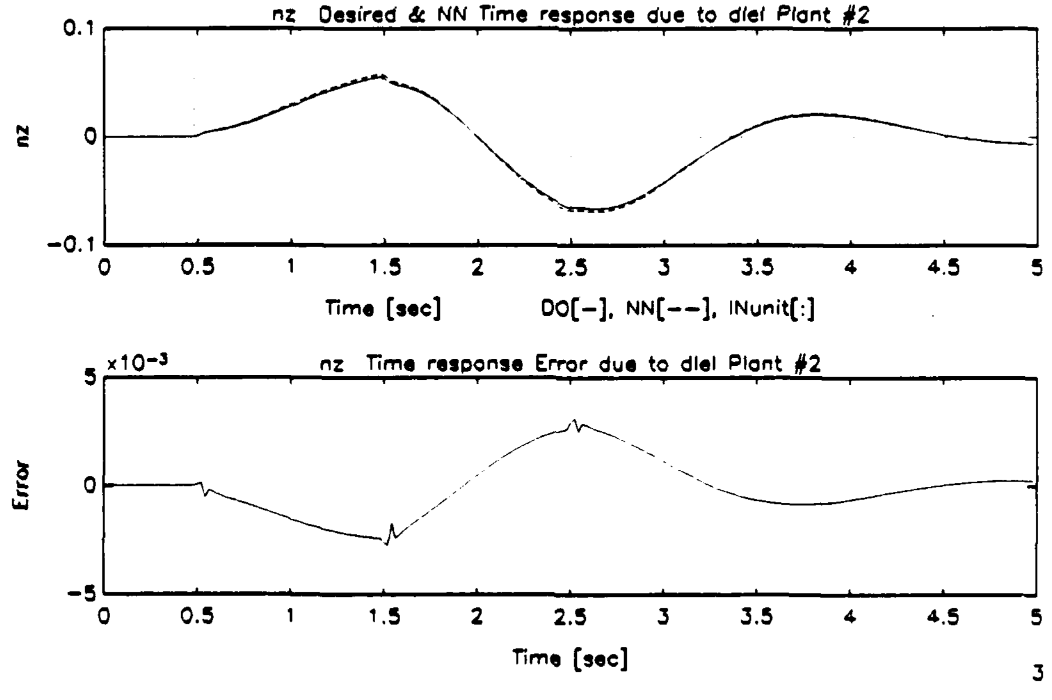
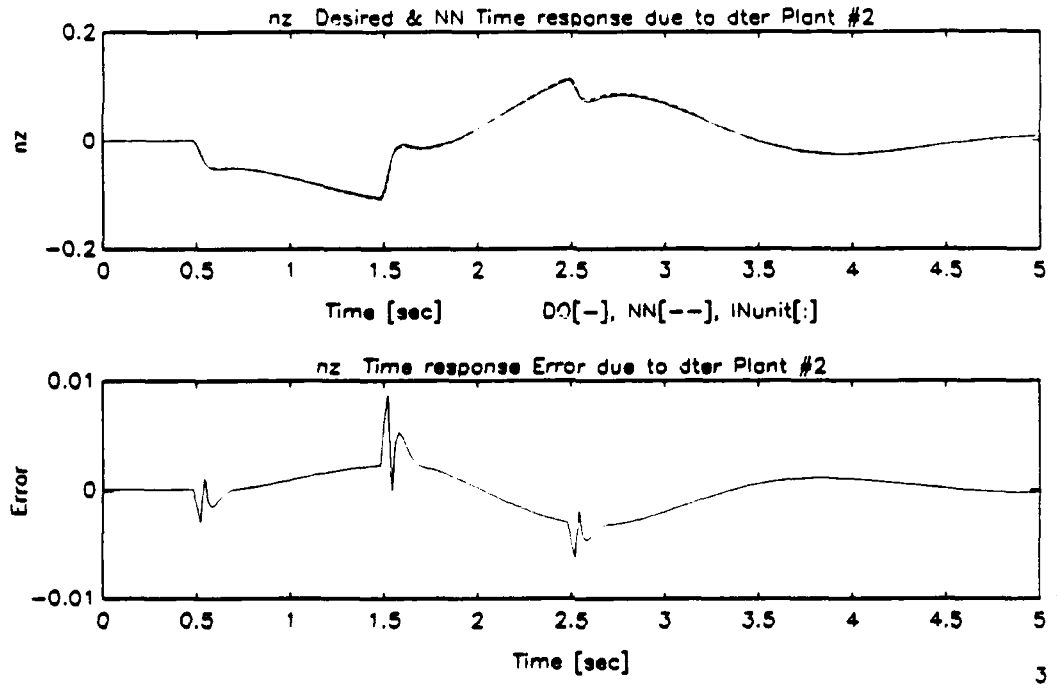


Figure B.36: **Normal Acceleration Response Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



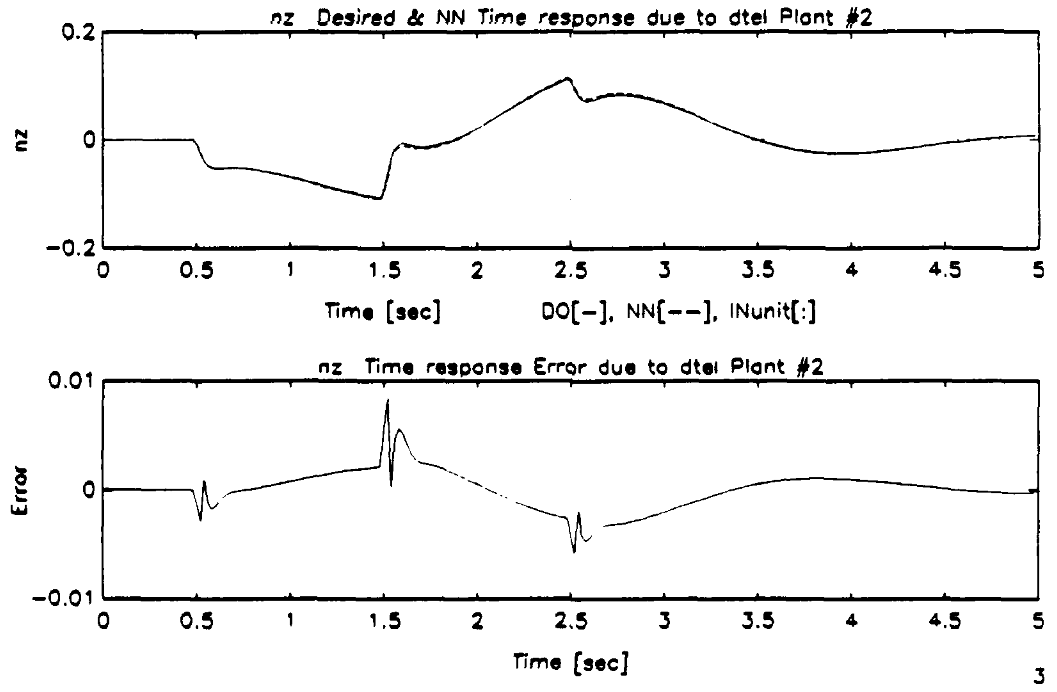
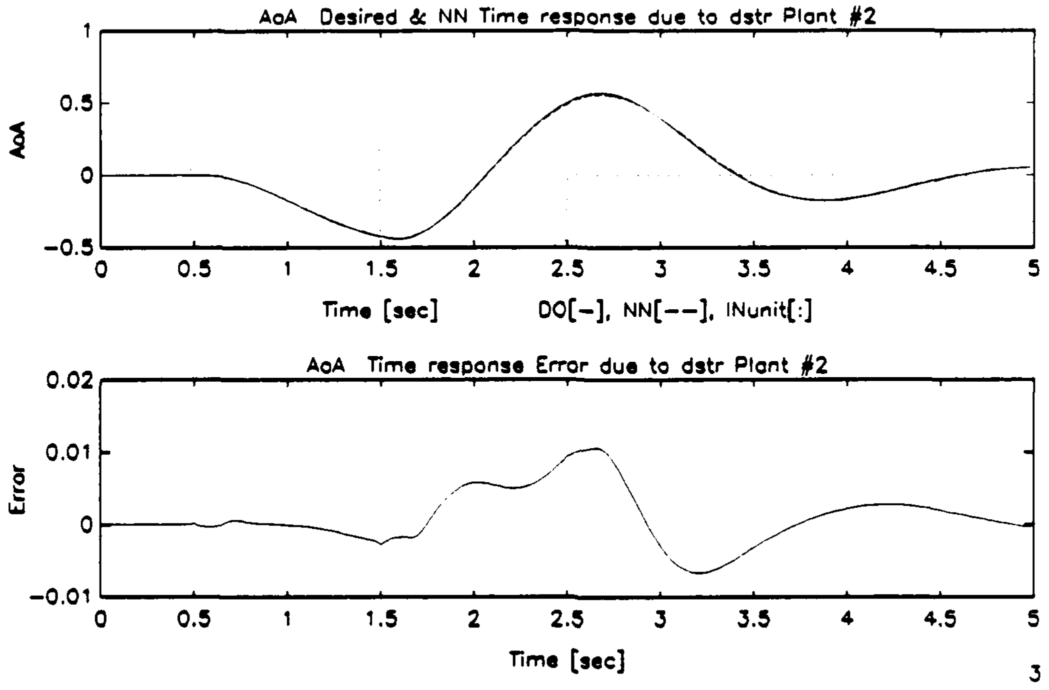


Figure B.37: **Normal Acceleration Response Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**



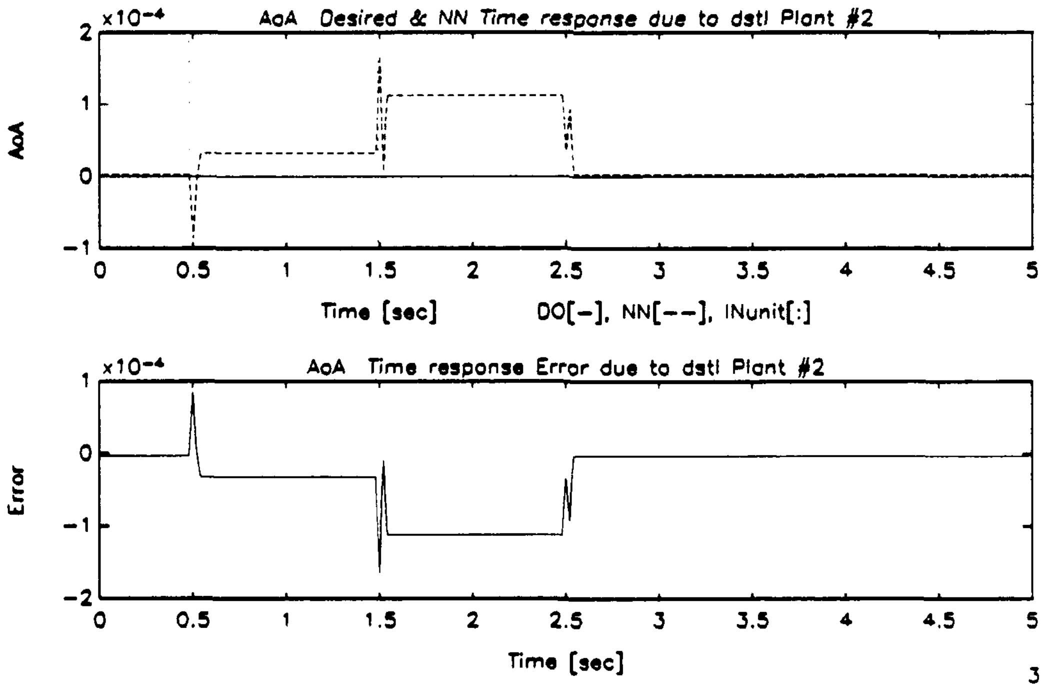


Figure B.38: **Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Single Network Emulation**

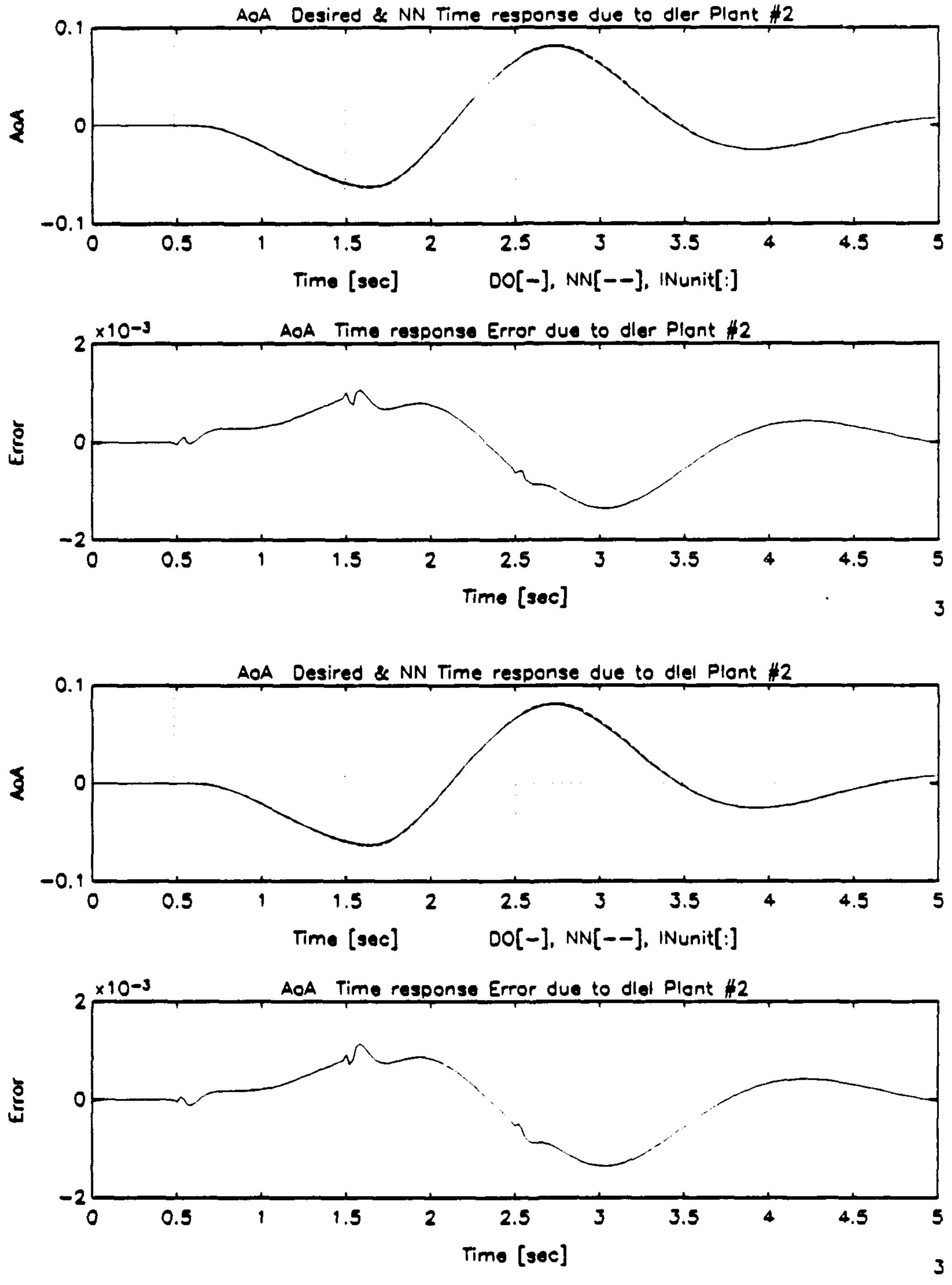
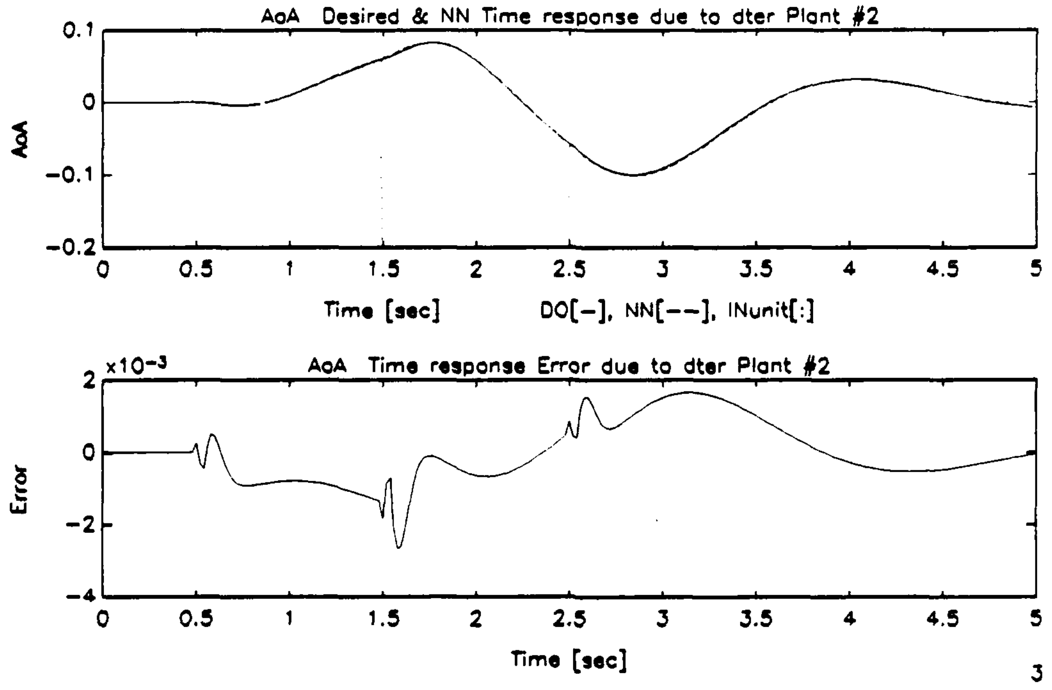


Figure B.39: Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation



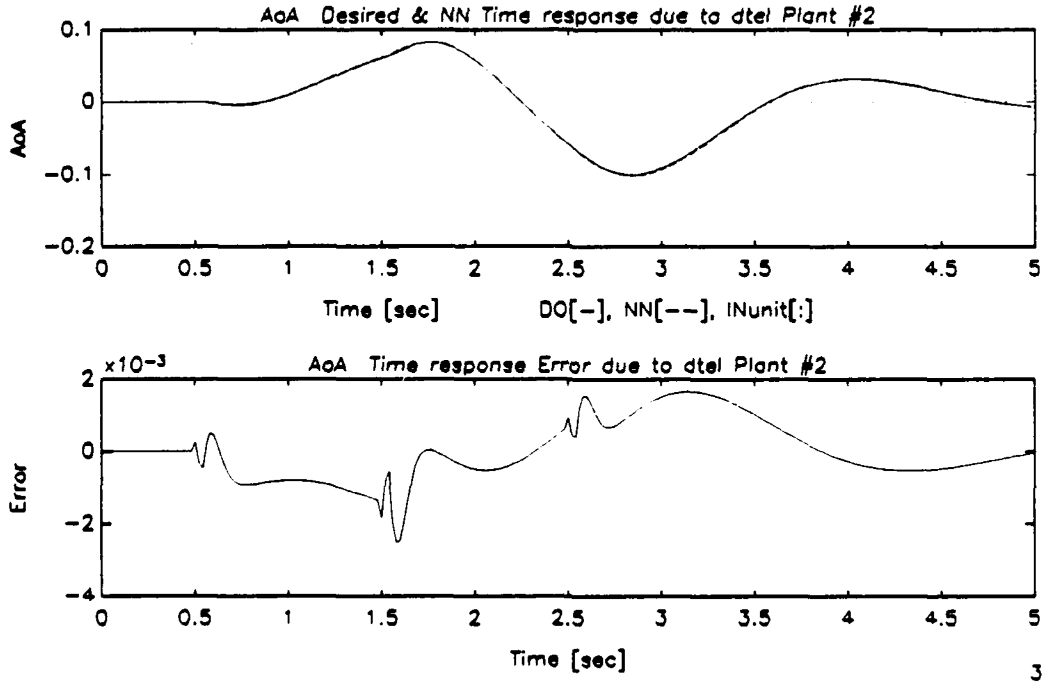


Figure B.40: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Single Network Emulation**

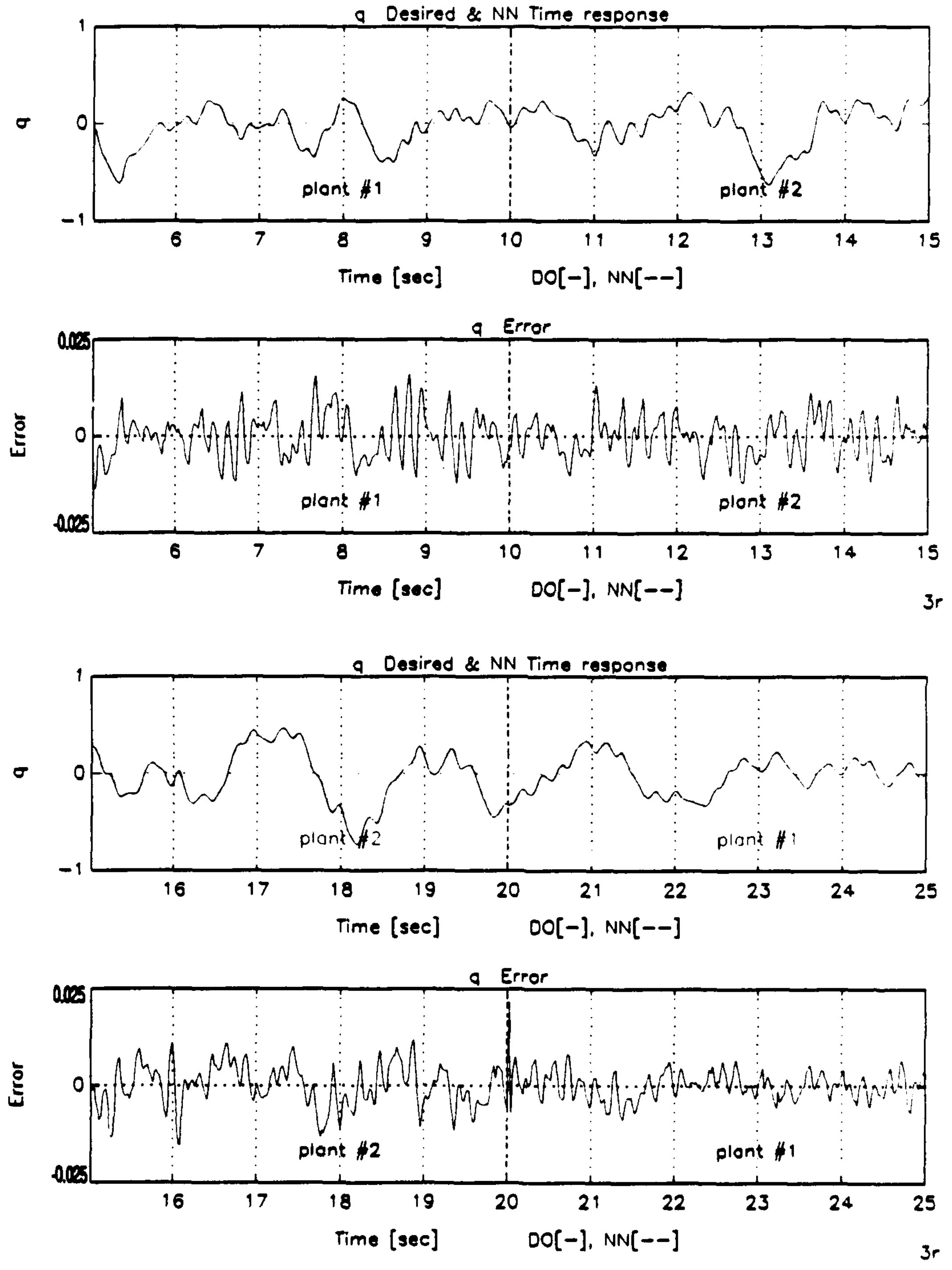
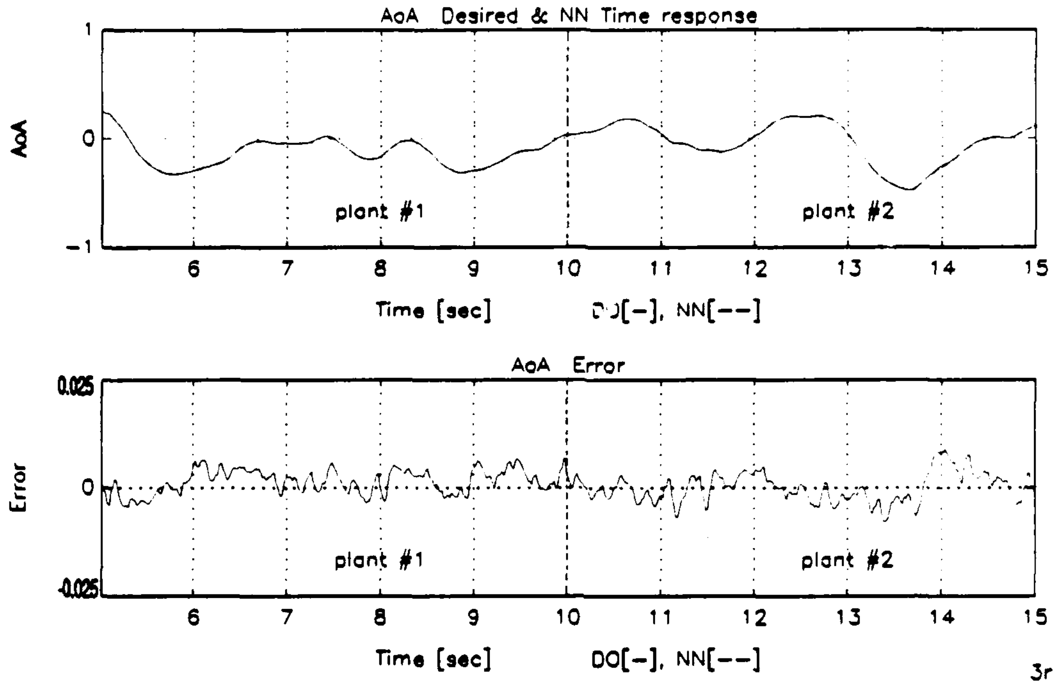


Figure B.41: Pitch Rate Response to RBS Excitation of All Inputs, Multiple-Plant, Single Network Emulation



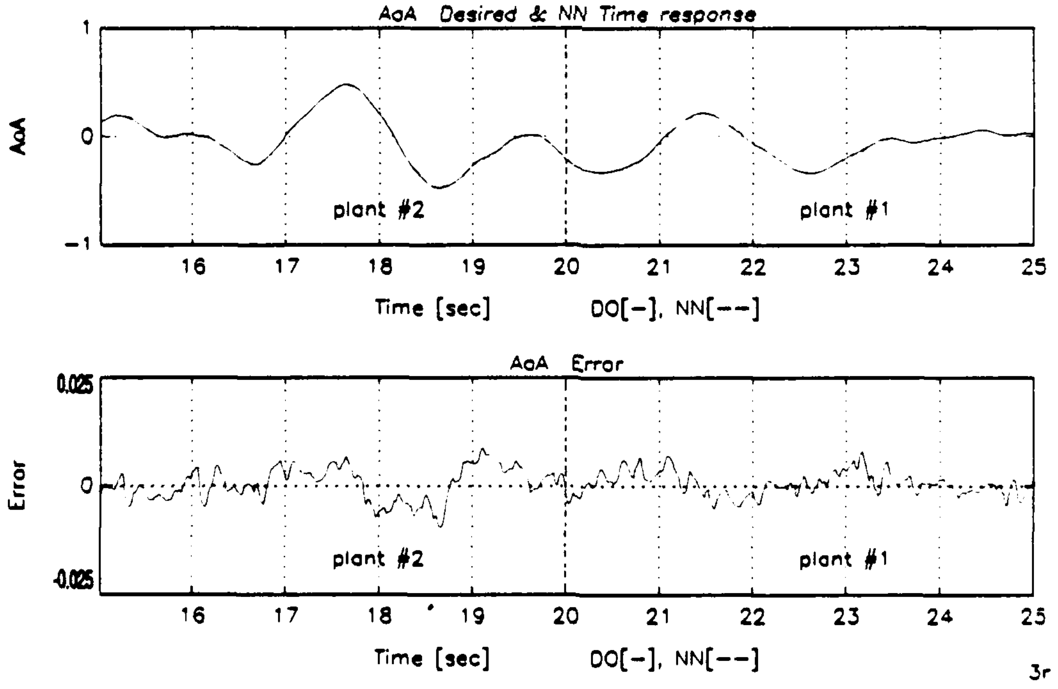


Figure B.42: **Angle of Attack Response to RBS Excitation of All Inputs, Multiple-Plant, Single Network Emulation**

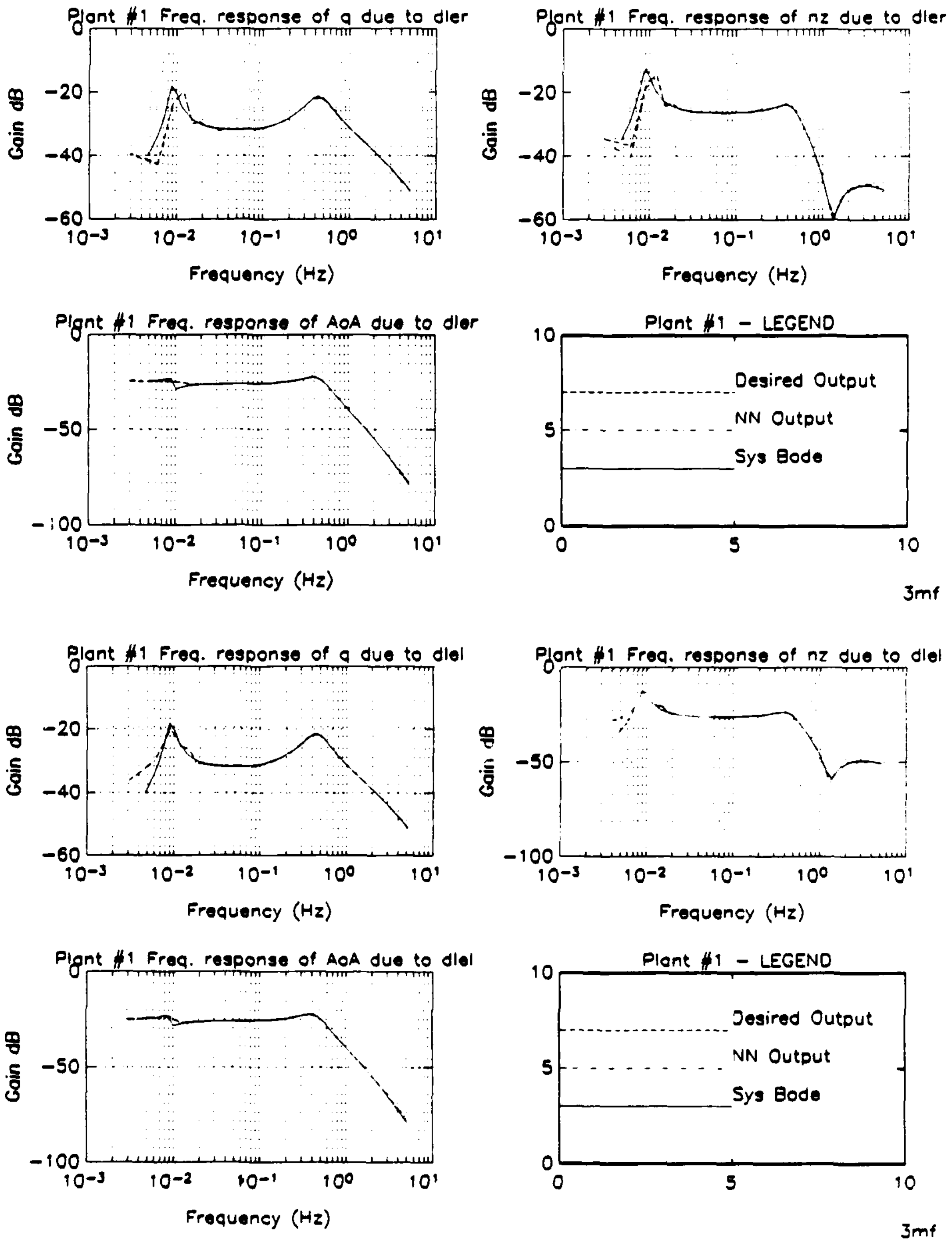


Figure B.43: **Spectral Response of Undamaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Minimized Network Emulation**

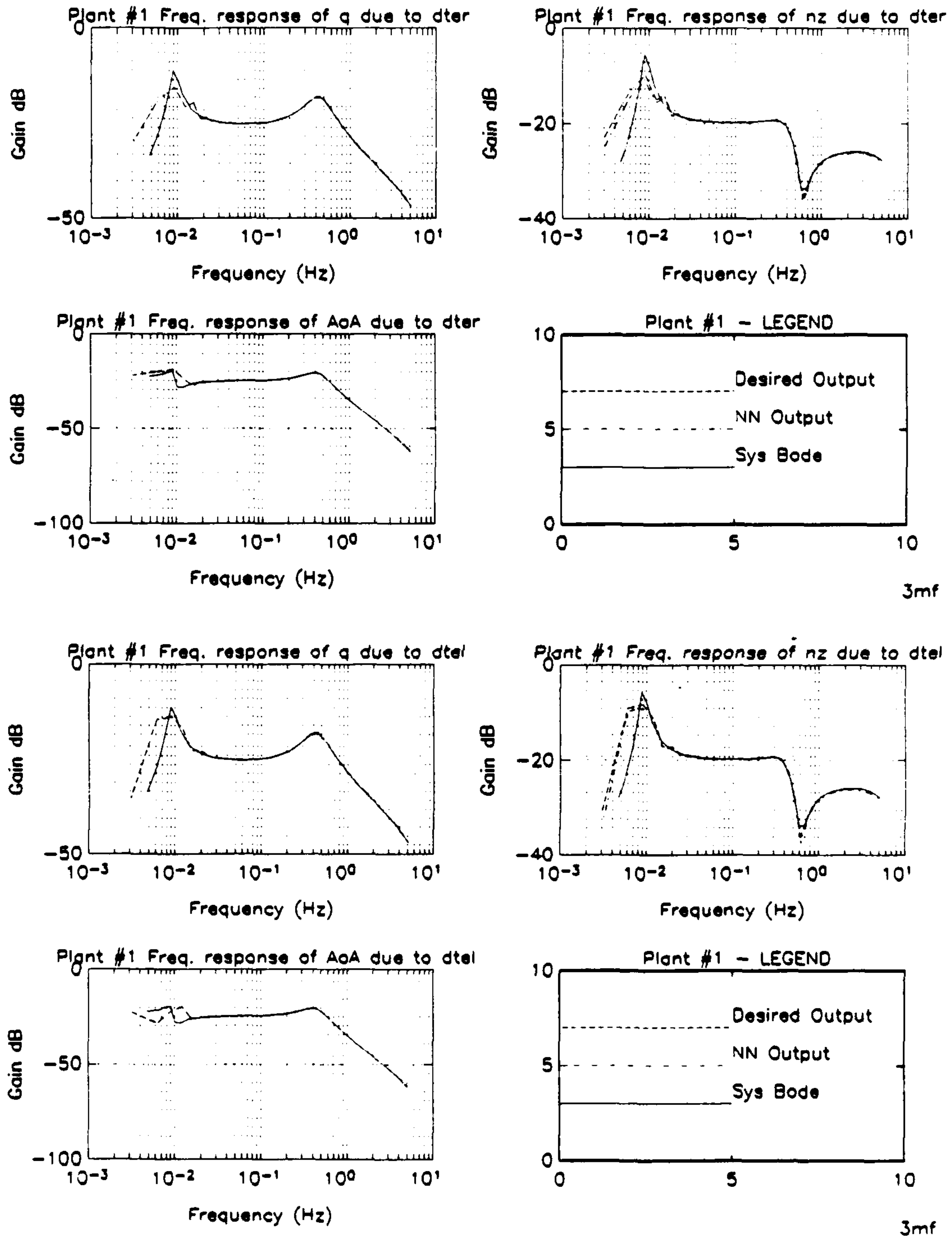


Figure B.44: **Spectral Response of Undamaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Minimized Network Emulation**

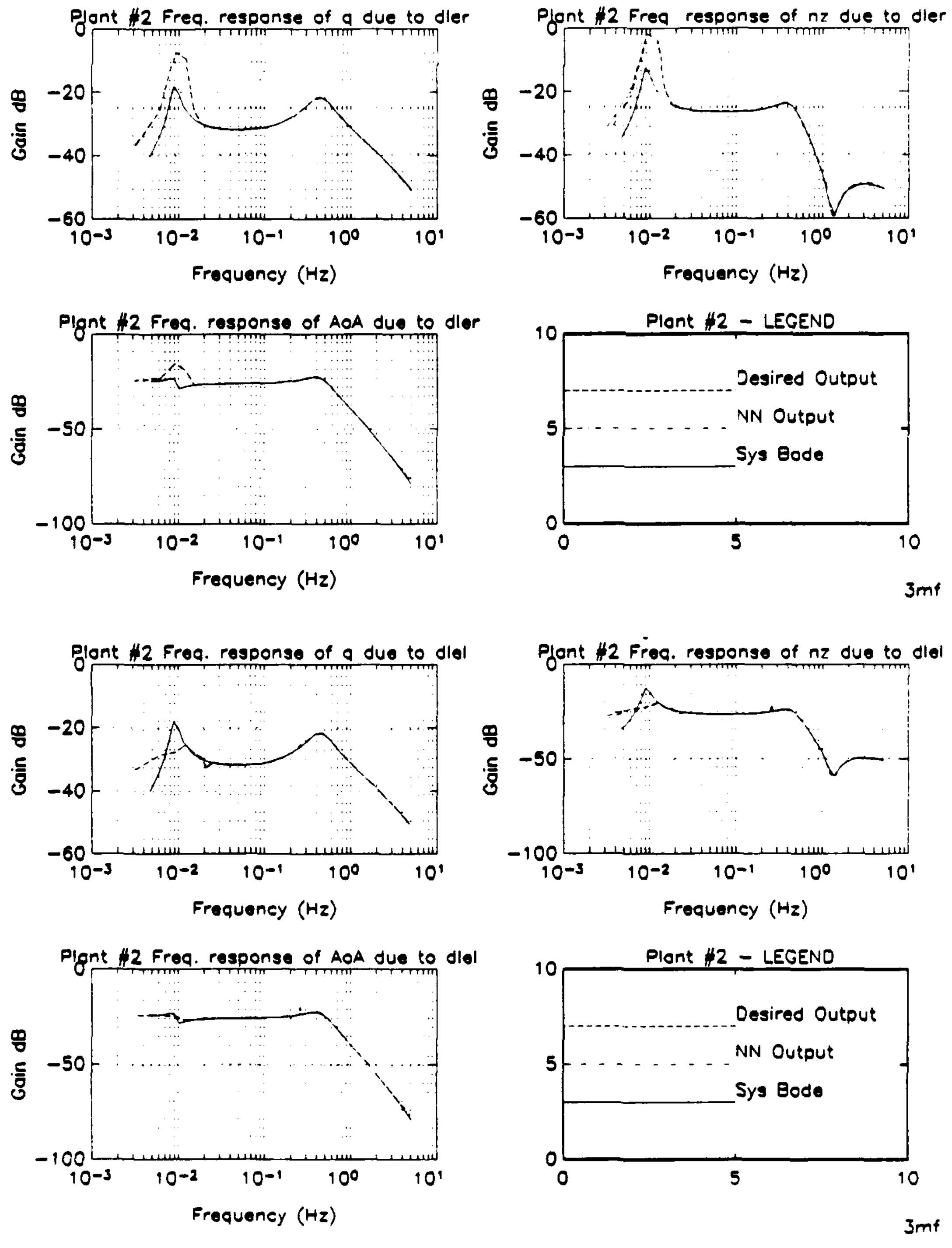


Figure B.45: **Spectral Response of Damaged Aircraft Due to Leading Edge Flaps, Multiple-Plant, Minimized Network Emulation**

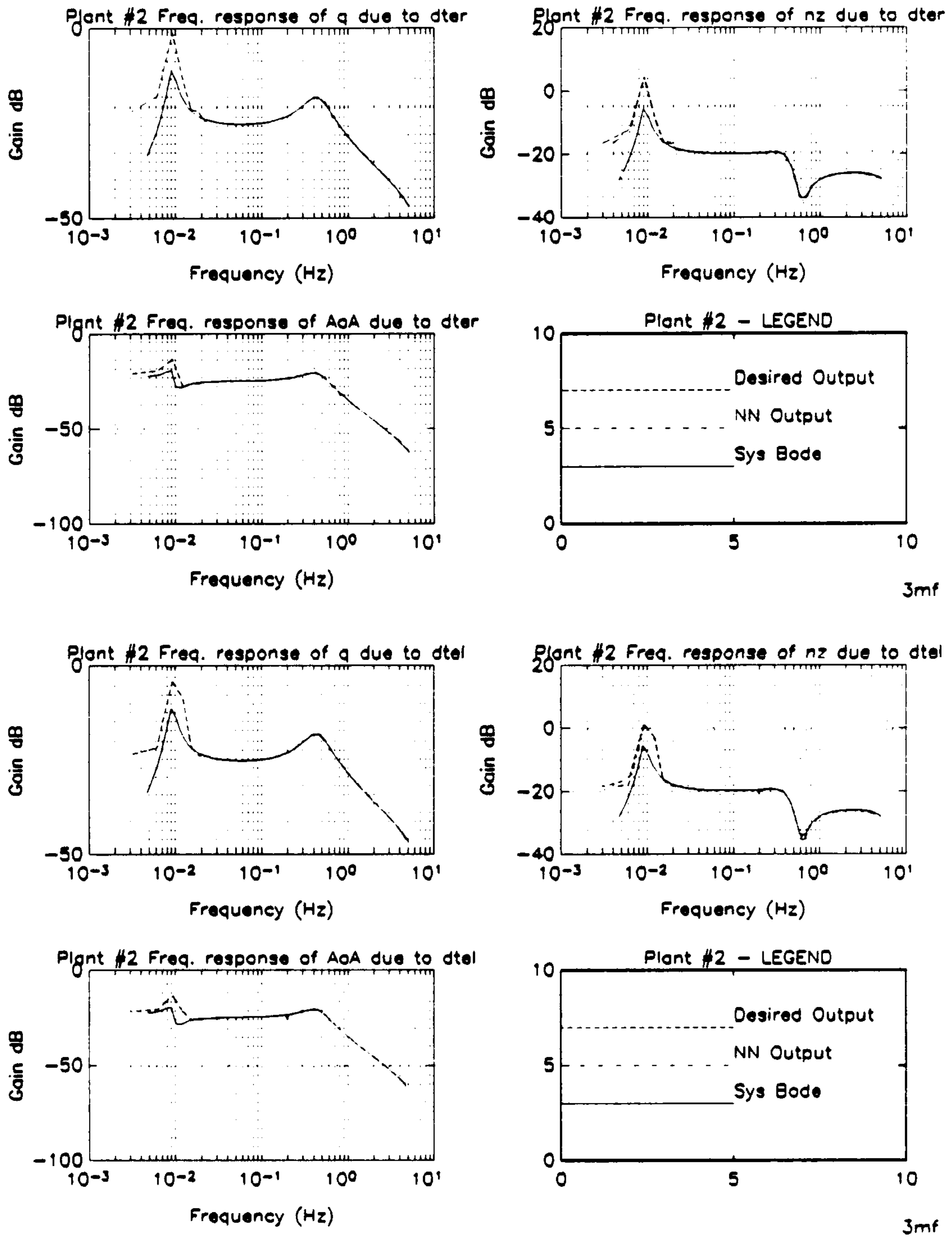


Figure B.46: **Spectral Response of Damaged Aircraft Due to Trailing Edge Flaps, Multiple-Plant, Minimized Network Emulation**

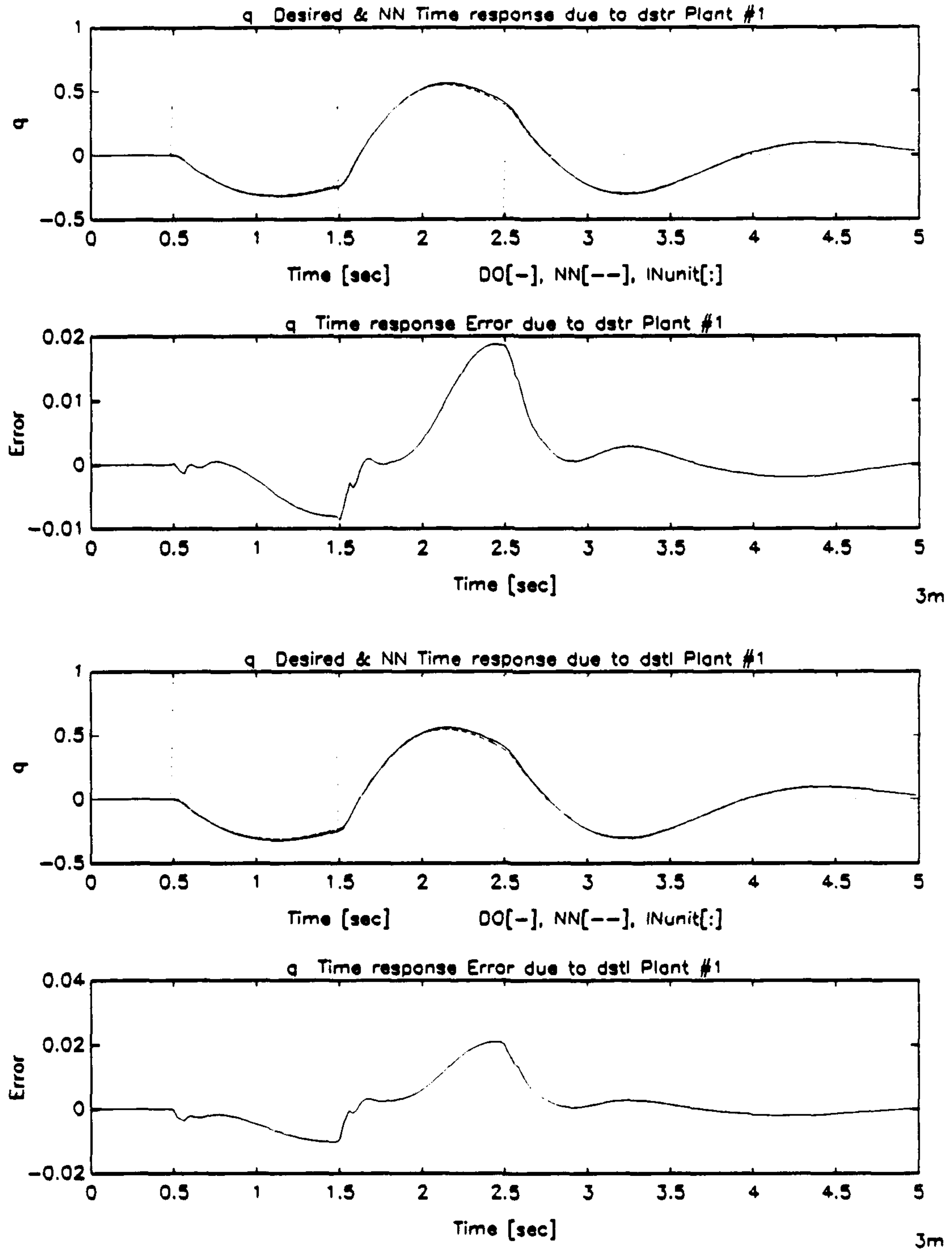


Figure B.47: **Pitch Rate Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation**

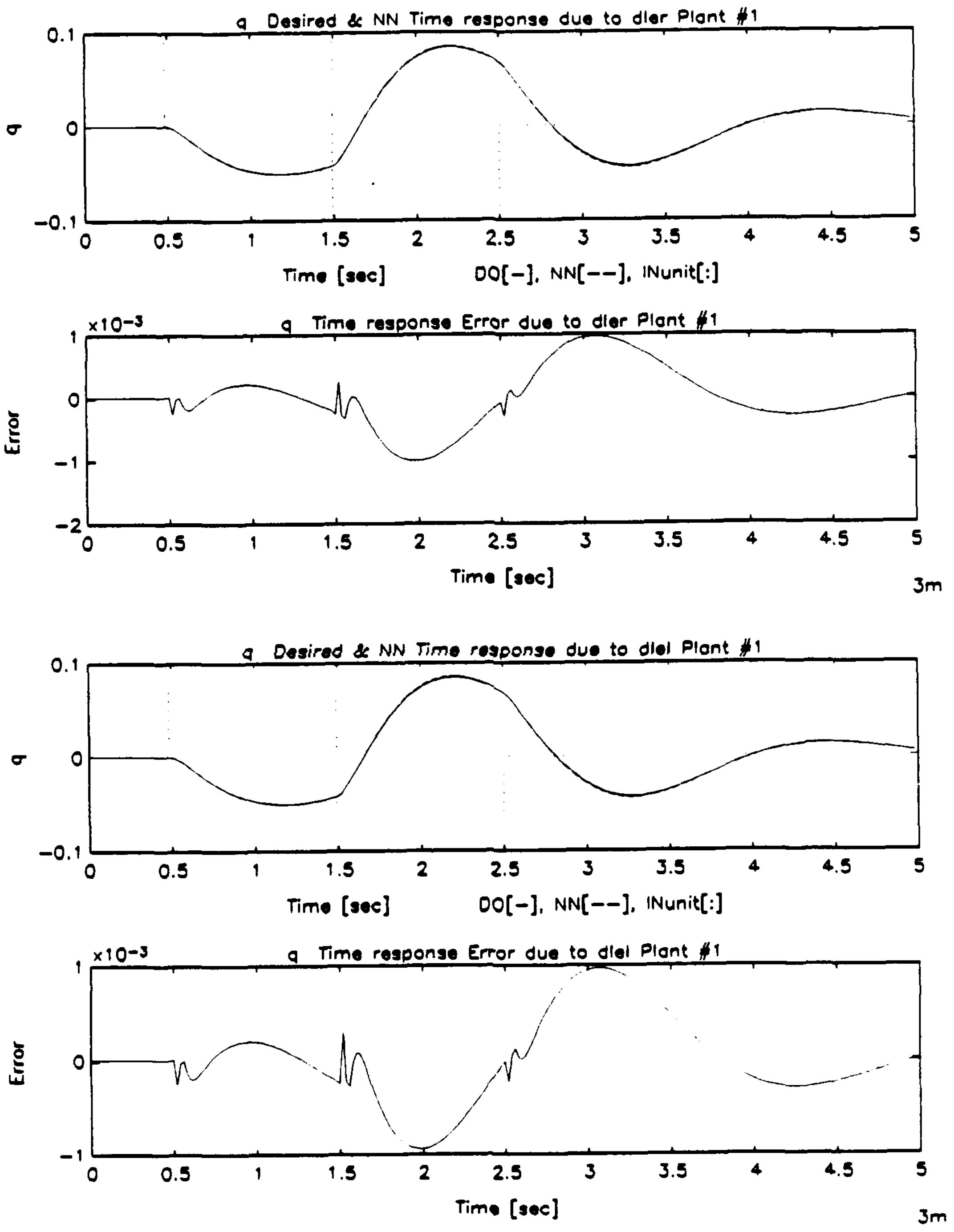
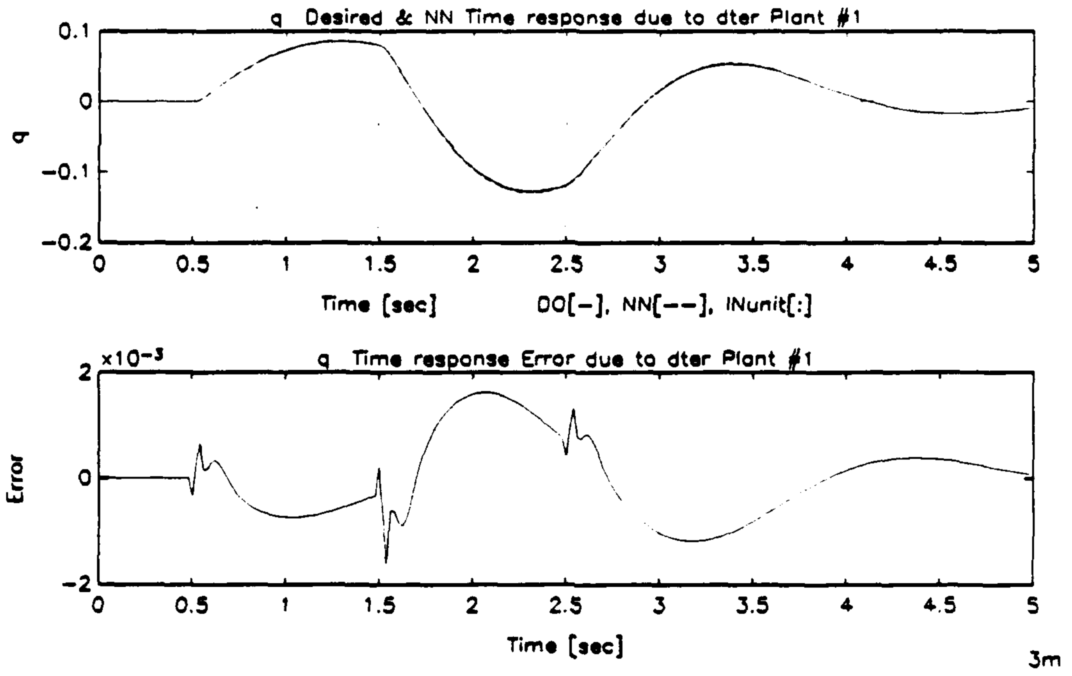


Figure B.48: **Pitch Rate Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**



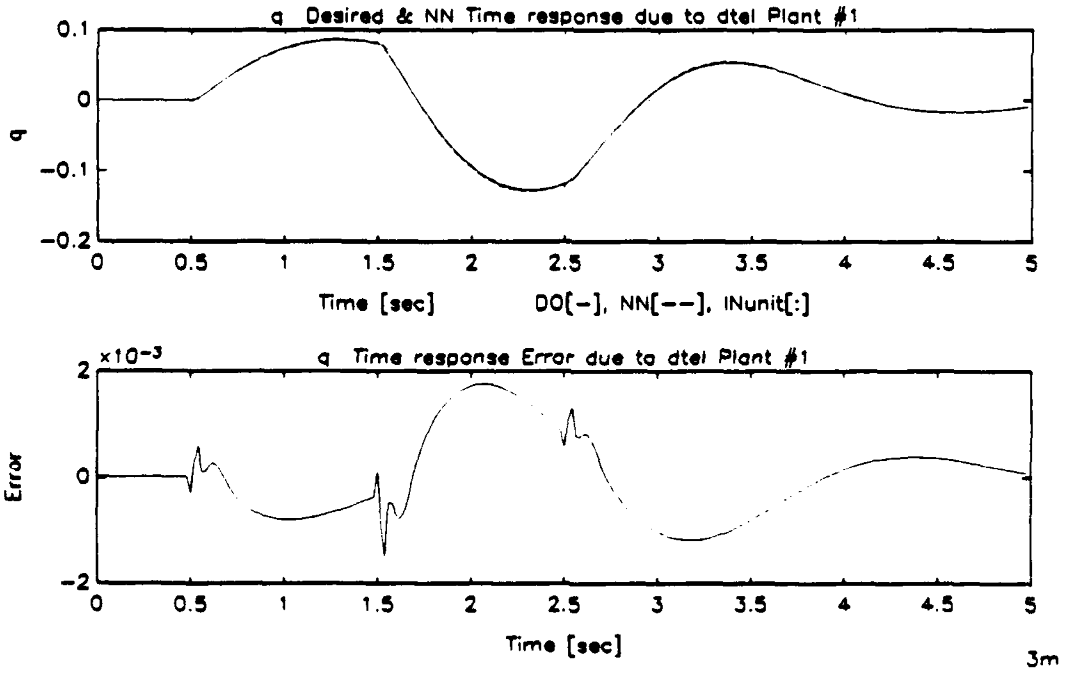


Figure B.49: **Pitch Rate Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

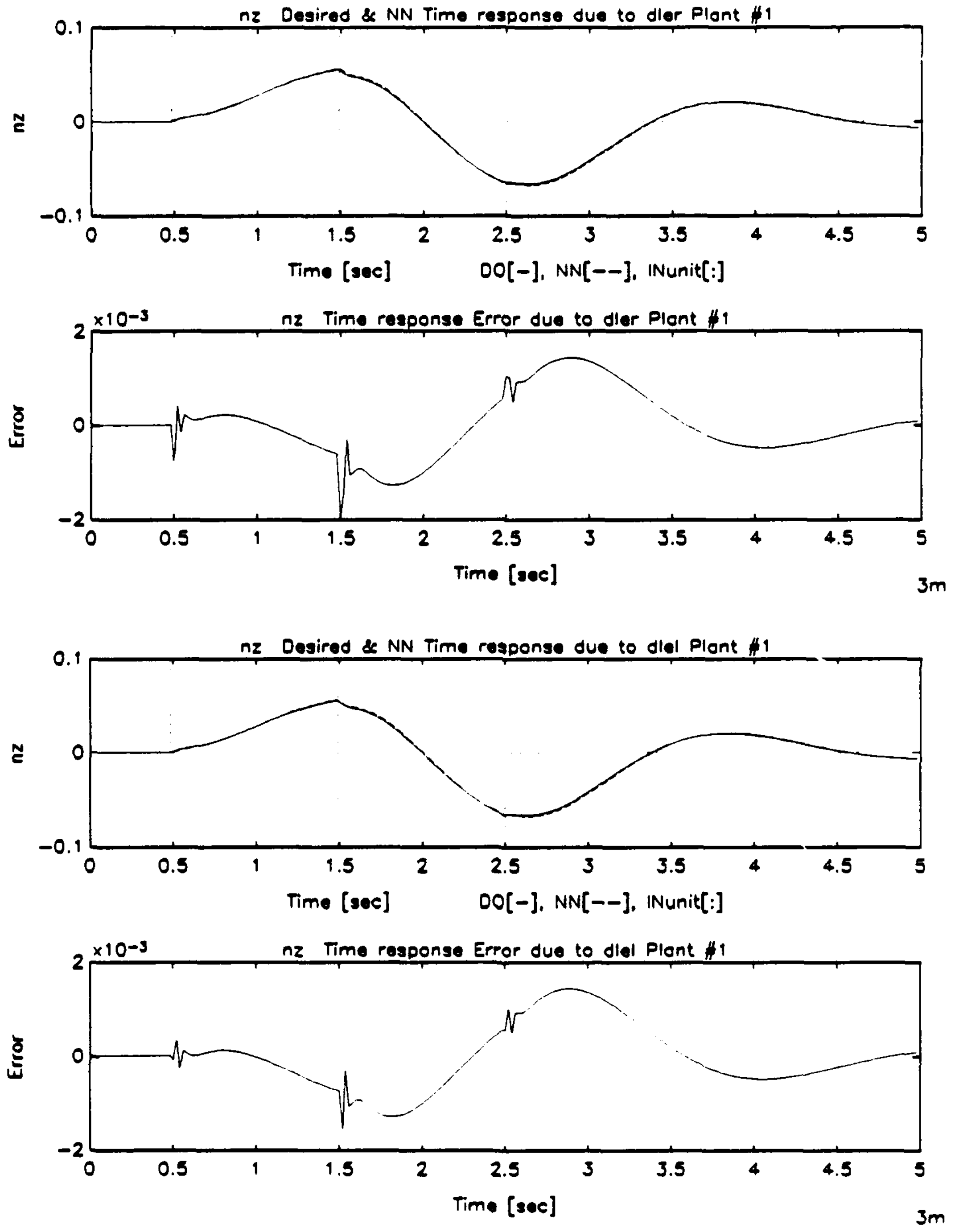


Figure B.50: Normal Acceleration Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation

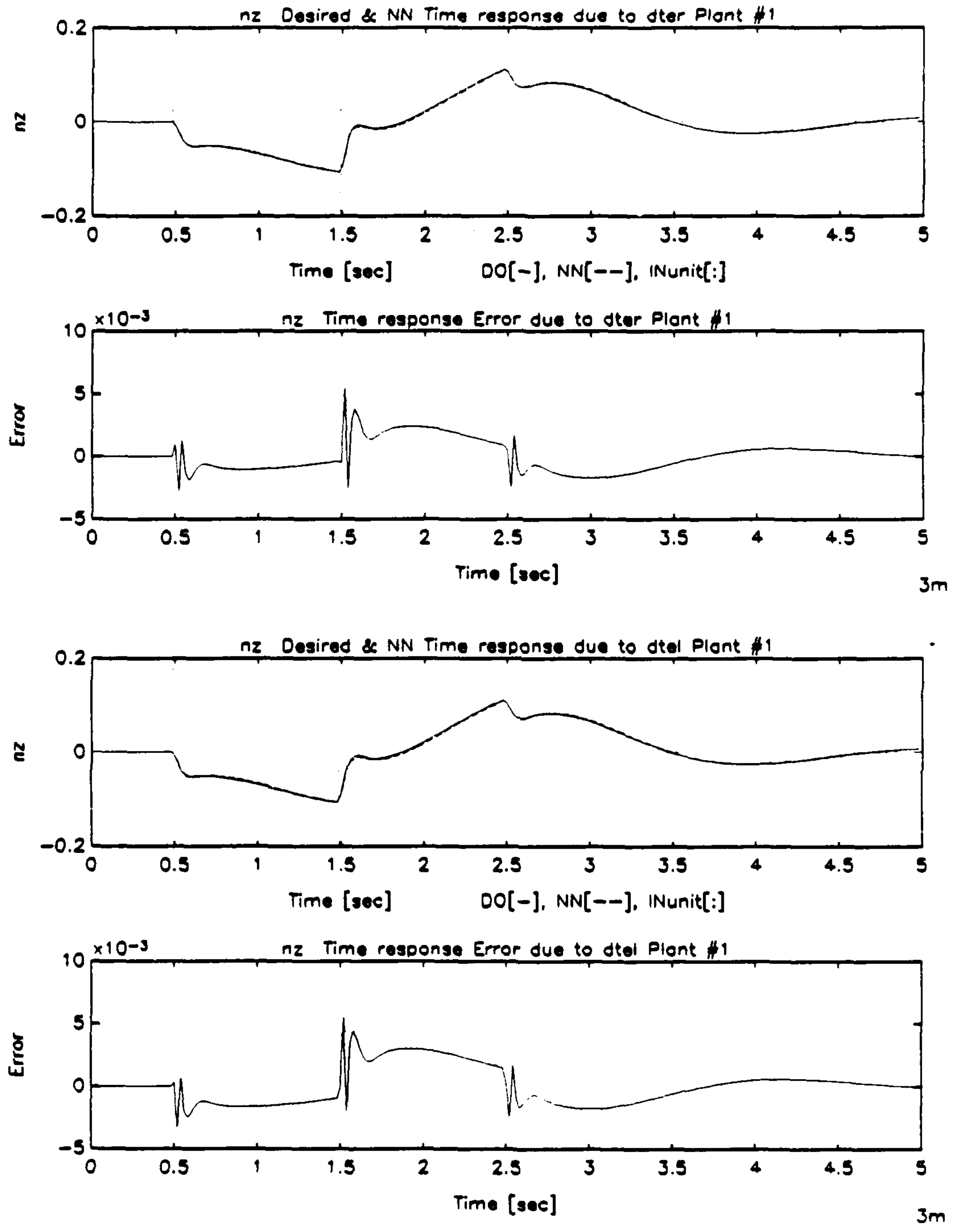


Figure B.51: Normal Acceleration Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation

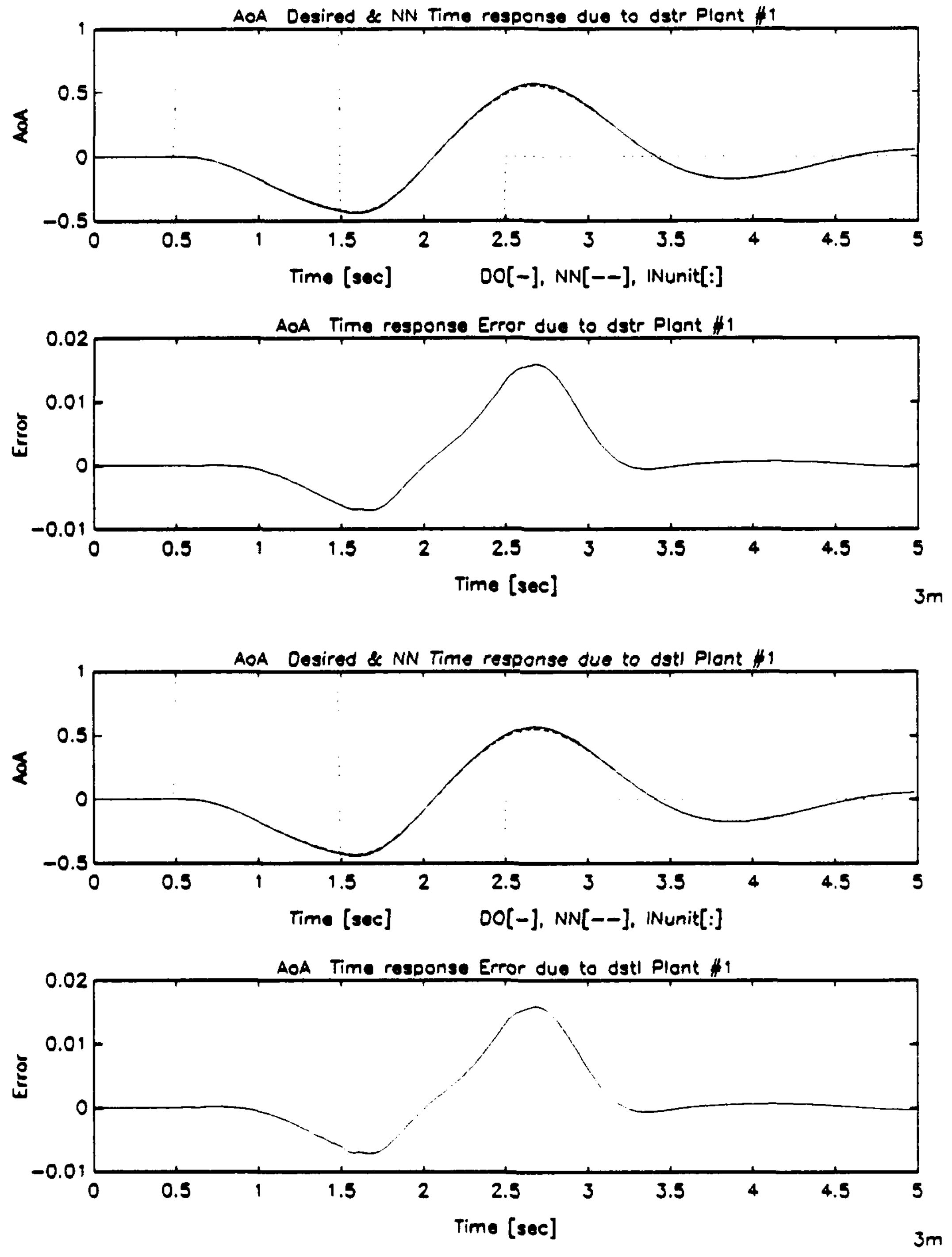


Figure B.52: Angle of Attack Response of Undamaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation

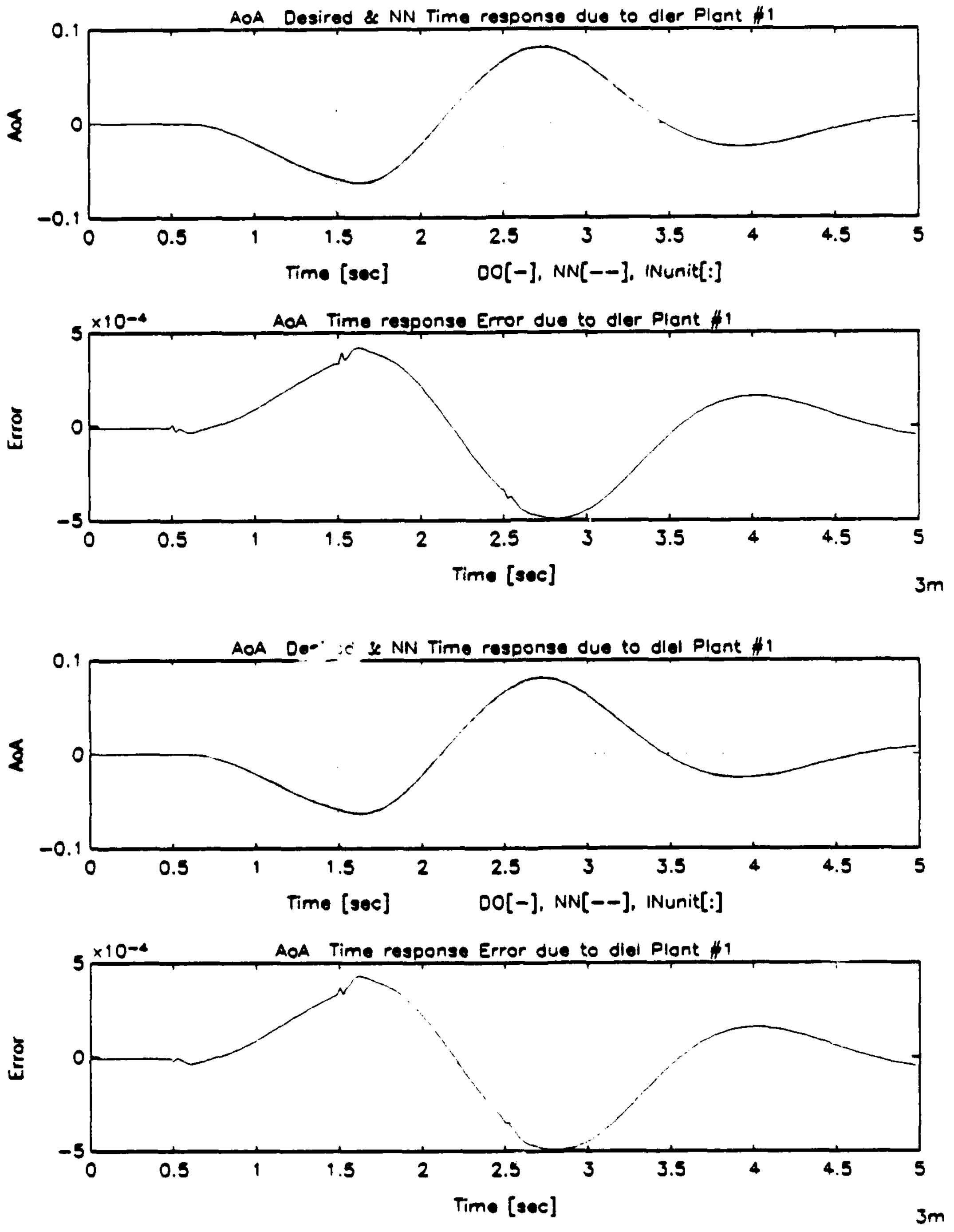
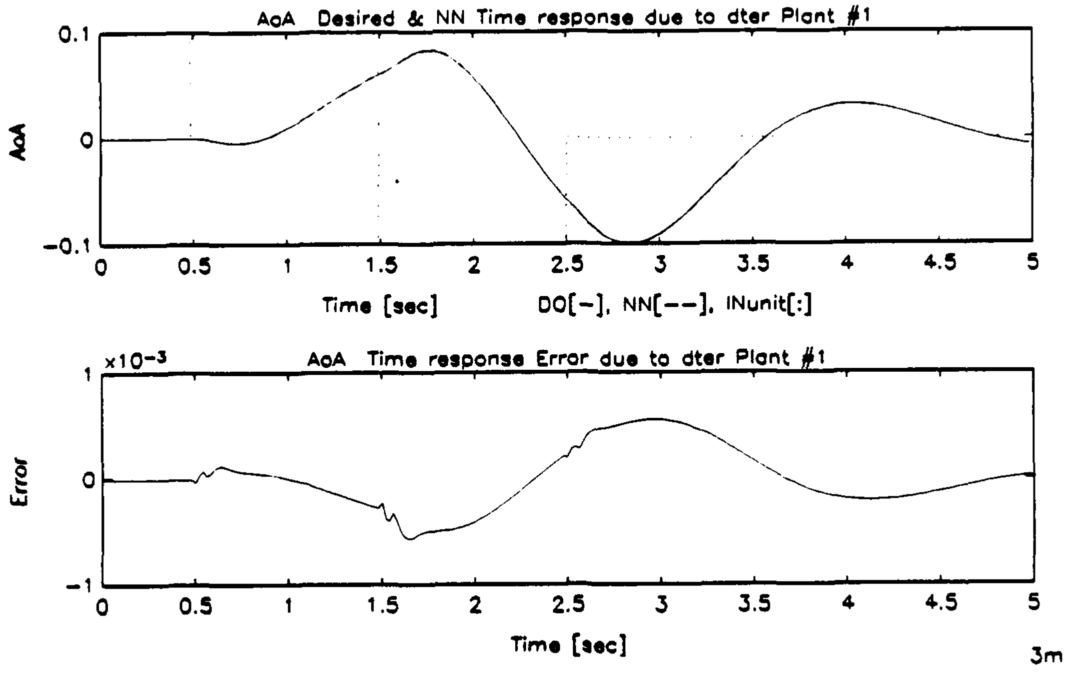


Figure B.53: **Angle of Attack Response of Undamaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**



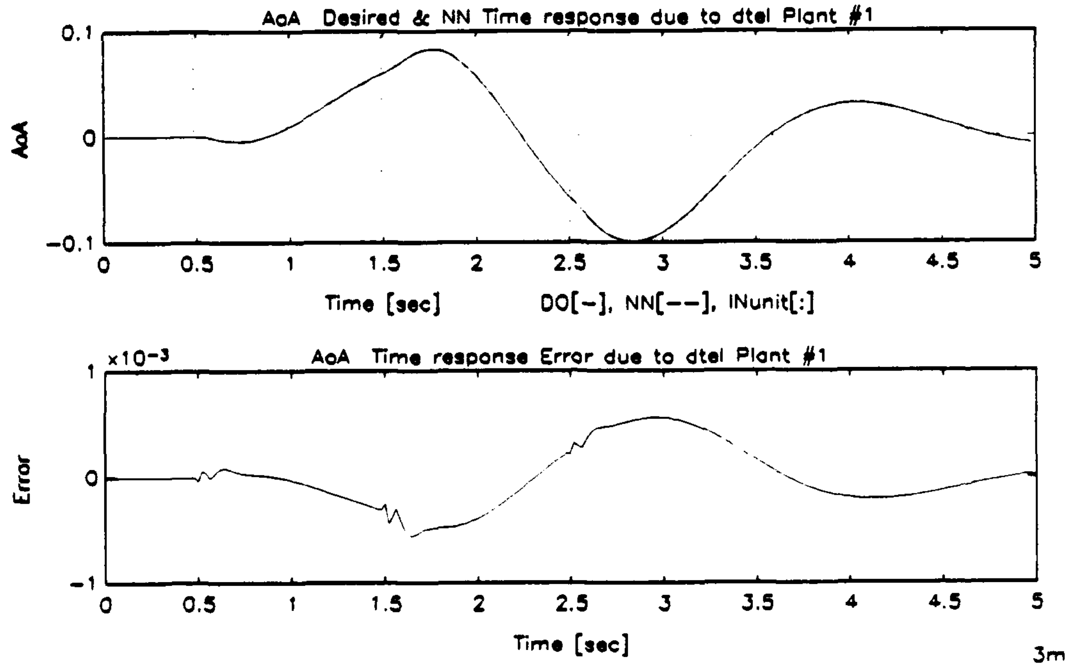


Figure B.54: **Angle of Attack Response of Undamaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

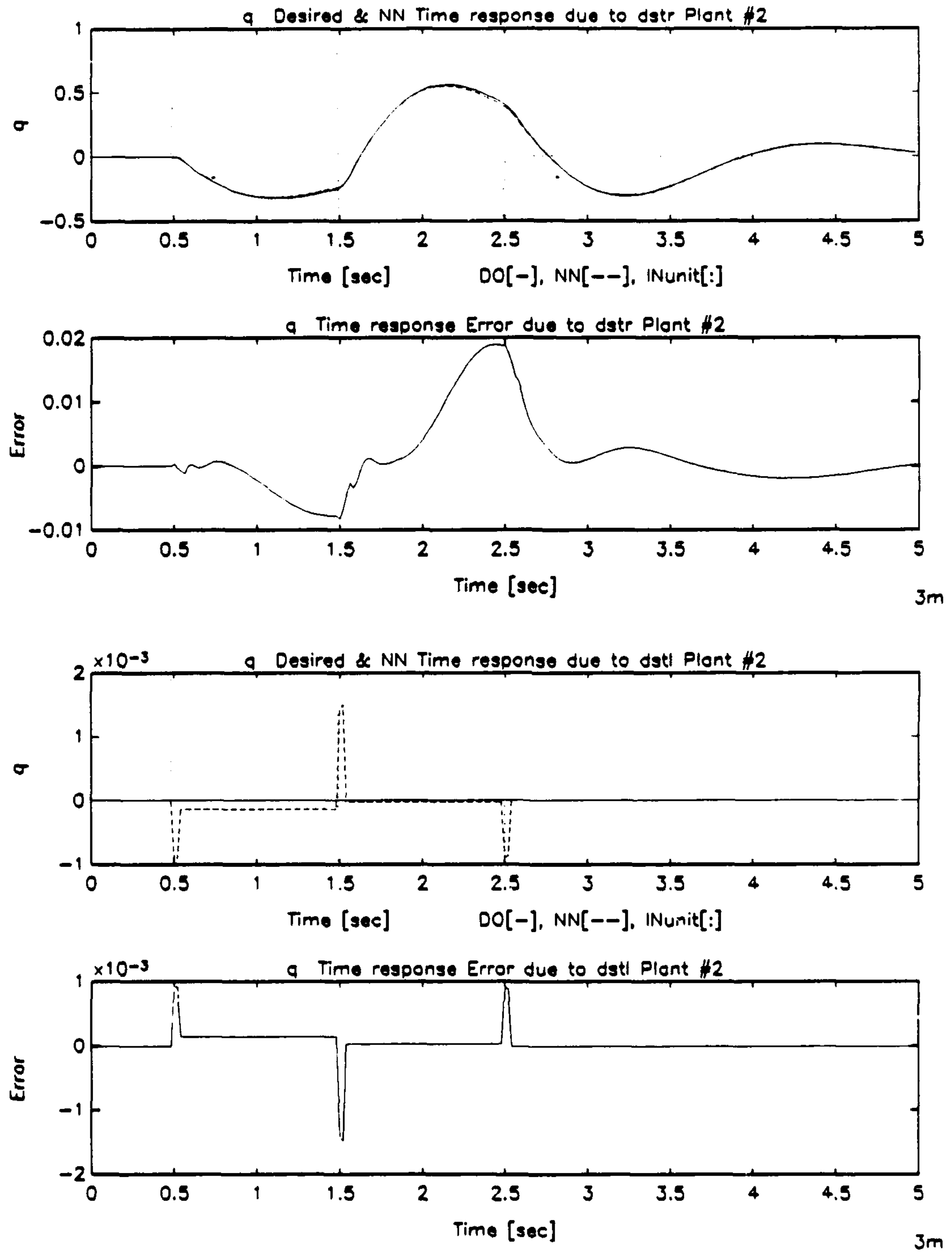


Figure B.55: **Pitch Rate Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation**

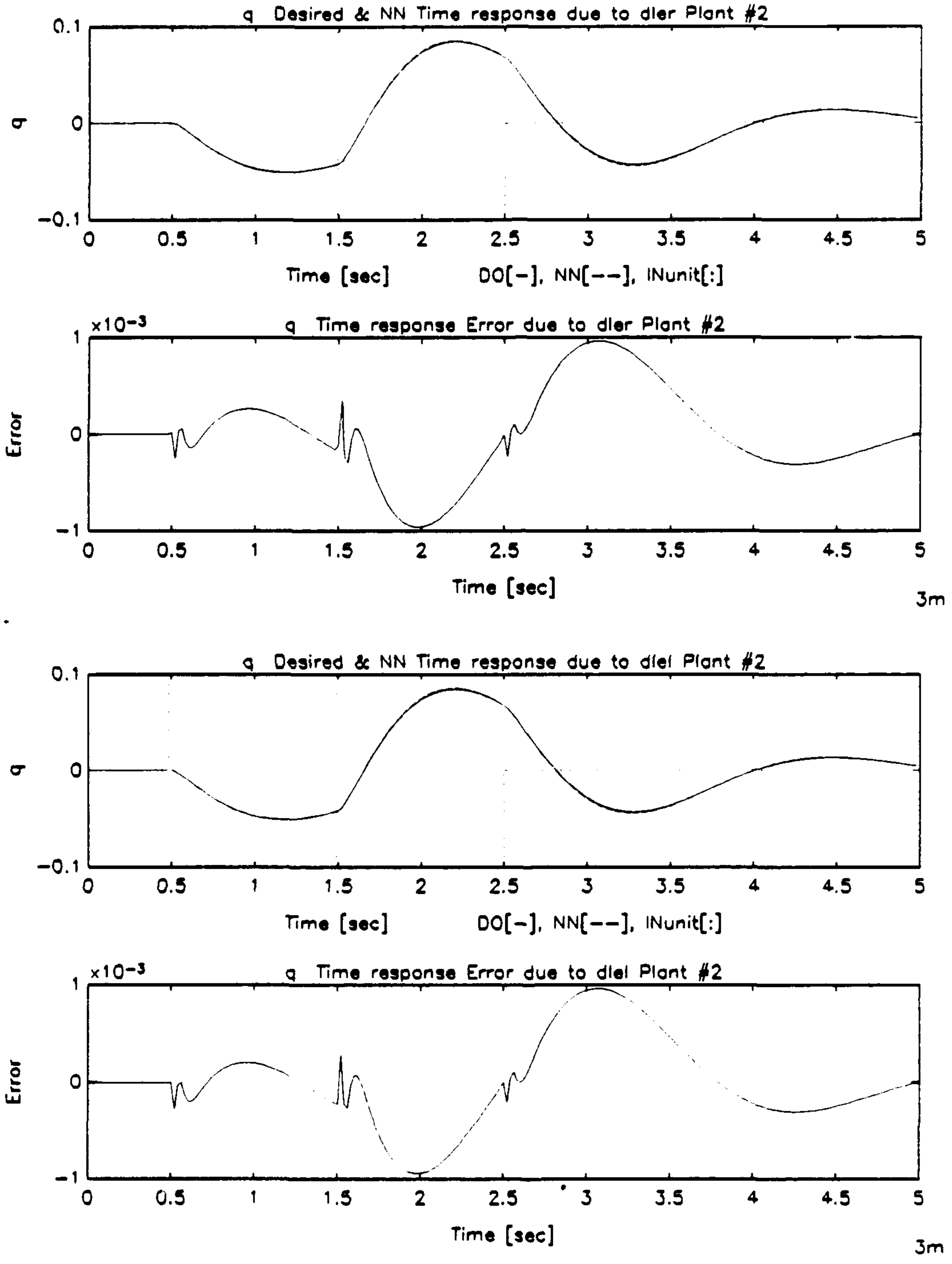


Figure B.56: **Pitch Rate Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

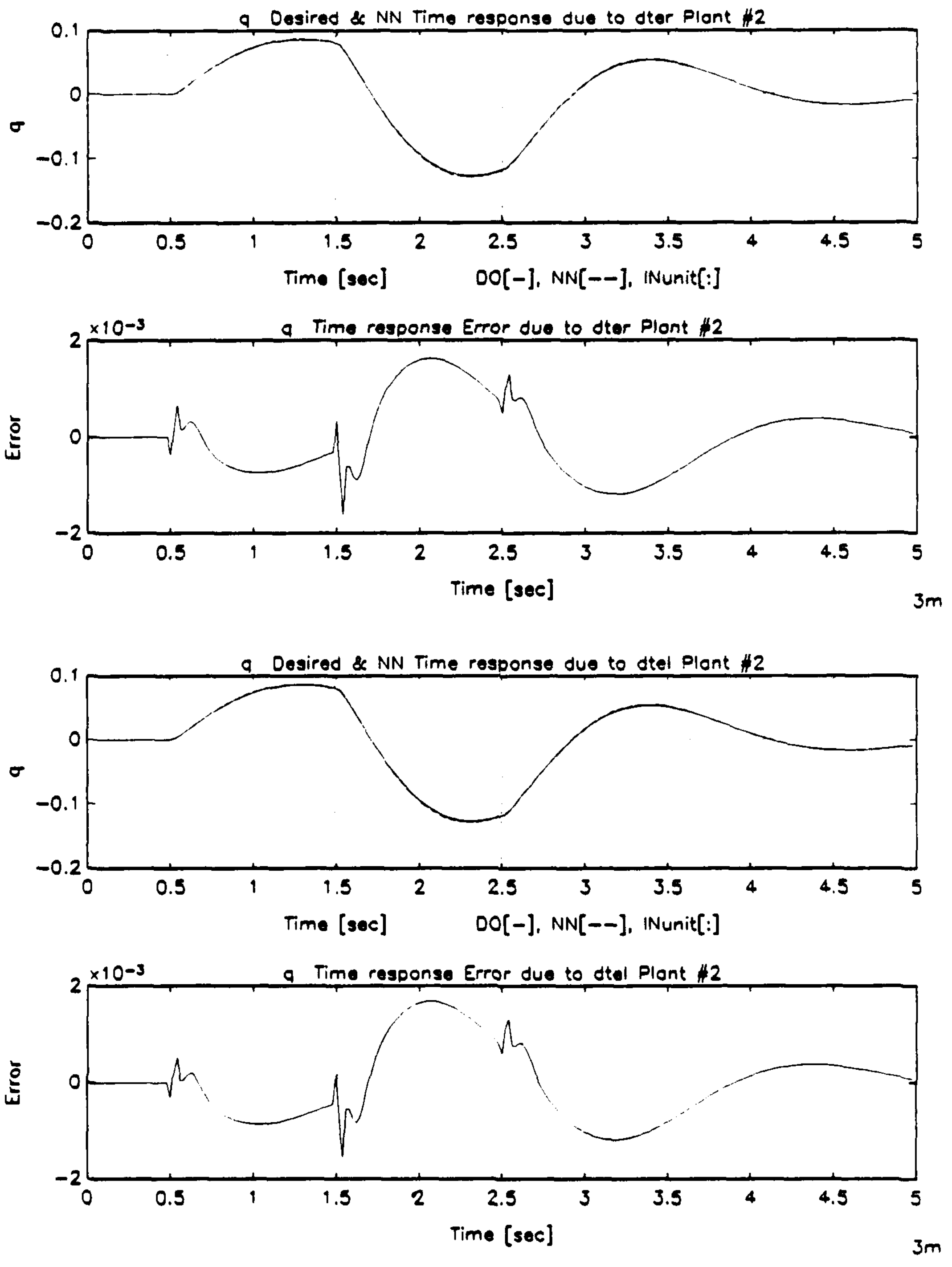


Figure B.57: **Pitch Rate Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

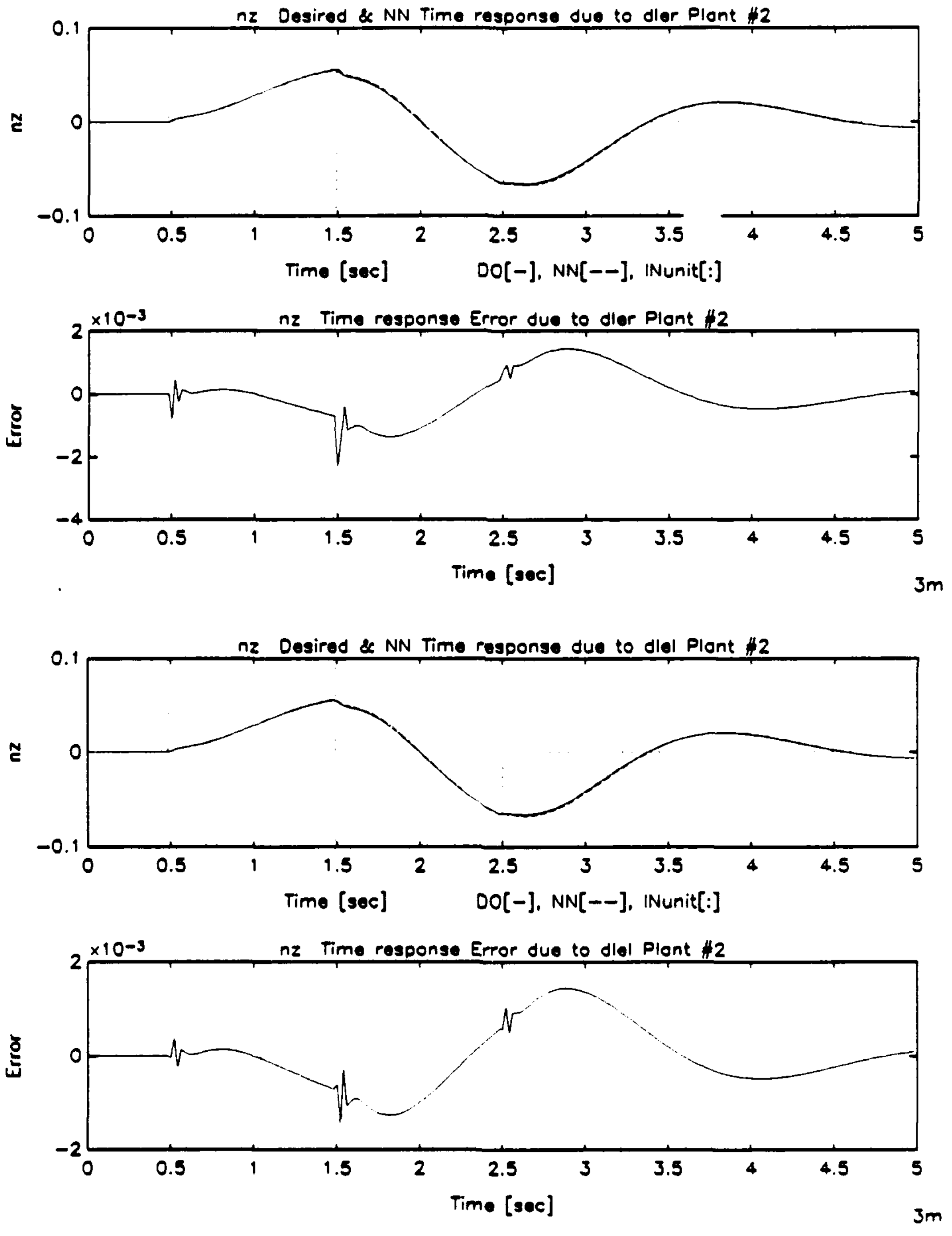


Figure B.58: **Normal Acceleration Response Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

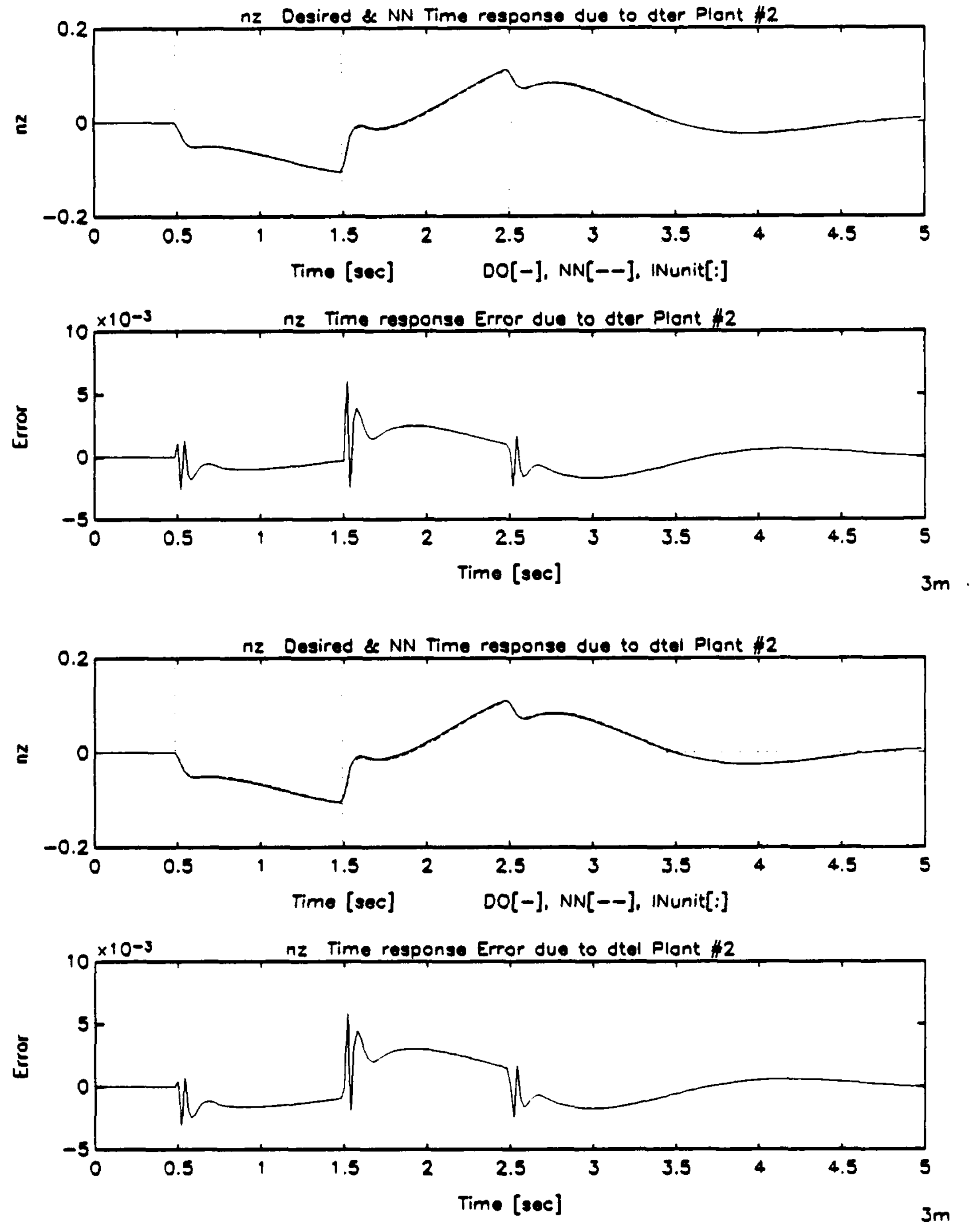
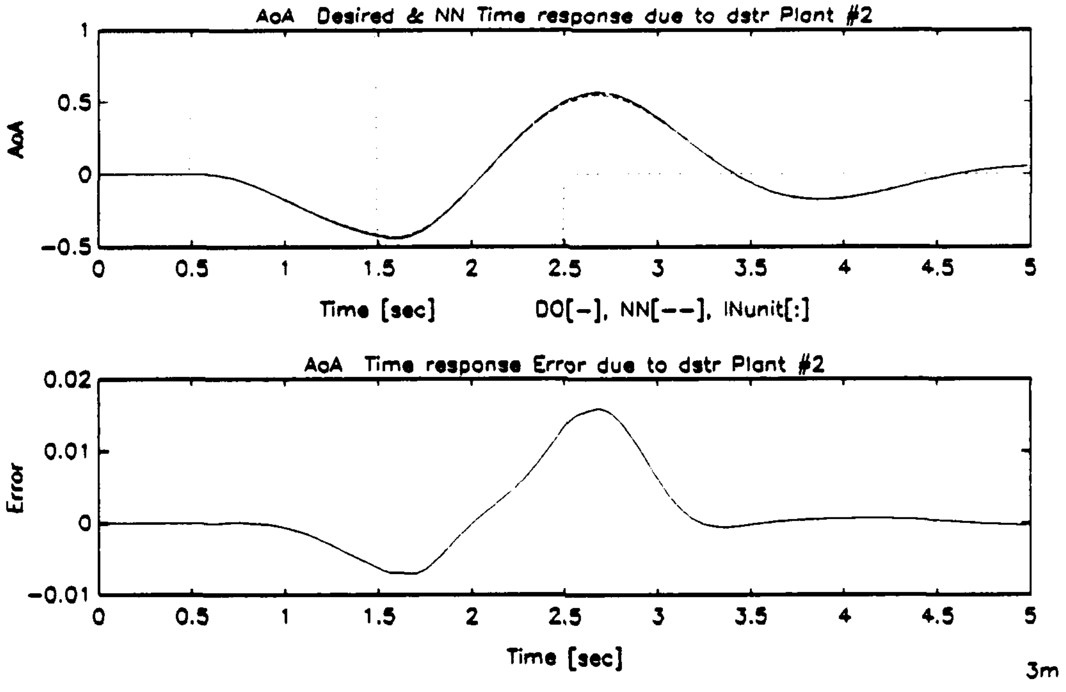


Figure B.59: **Normal Acceleration Response Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**



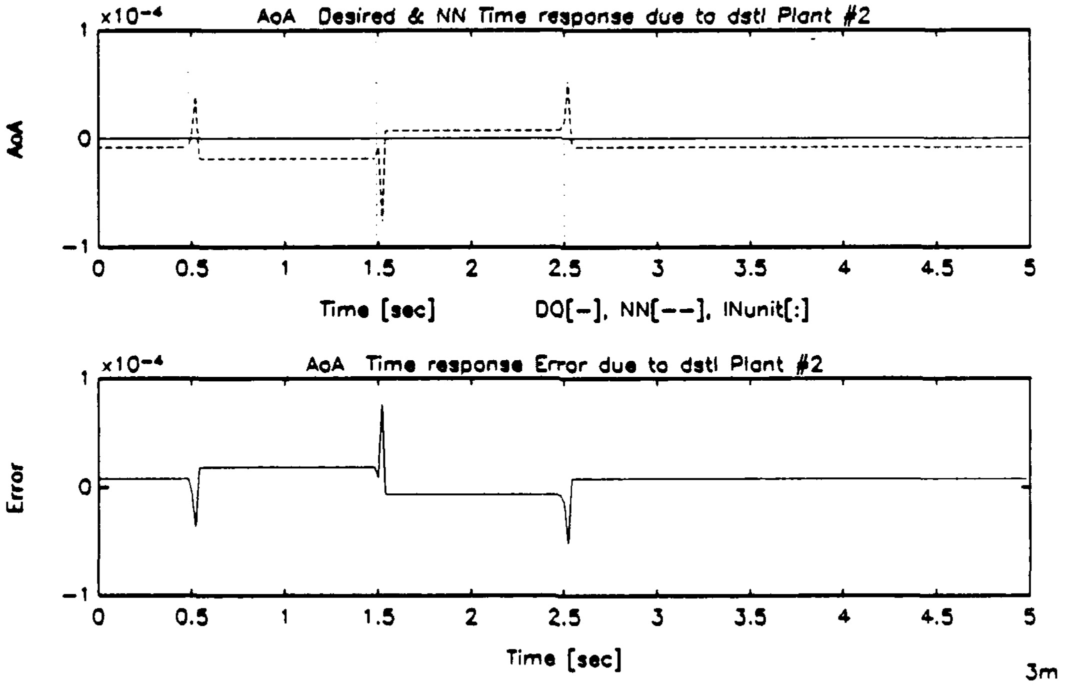
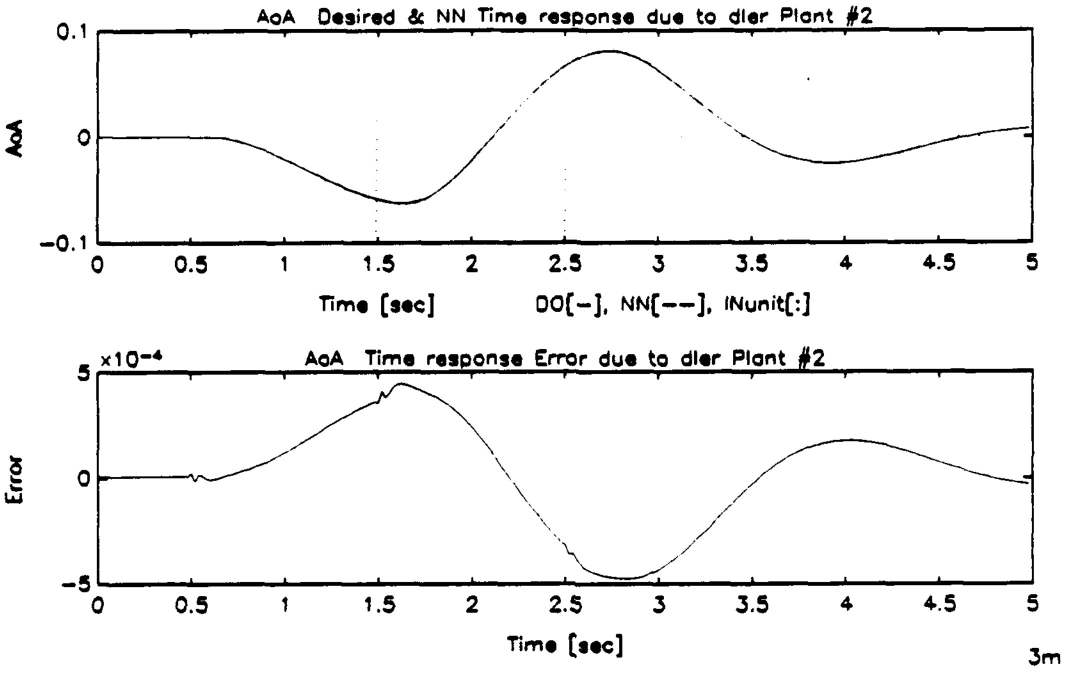


Figure B.60: Angle of Attack Response of Damaged Aircraft Due to Stabilators Dipole Command, Multiple-Plant, Minimized Network Emulation



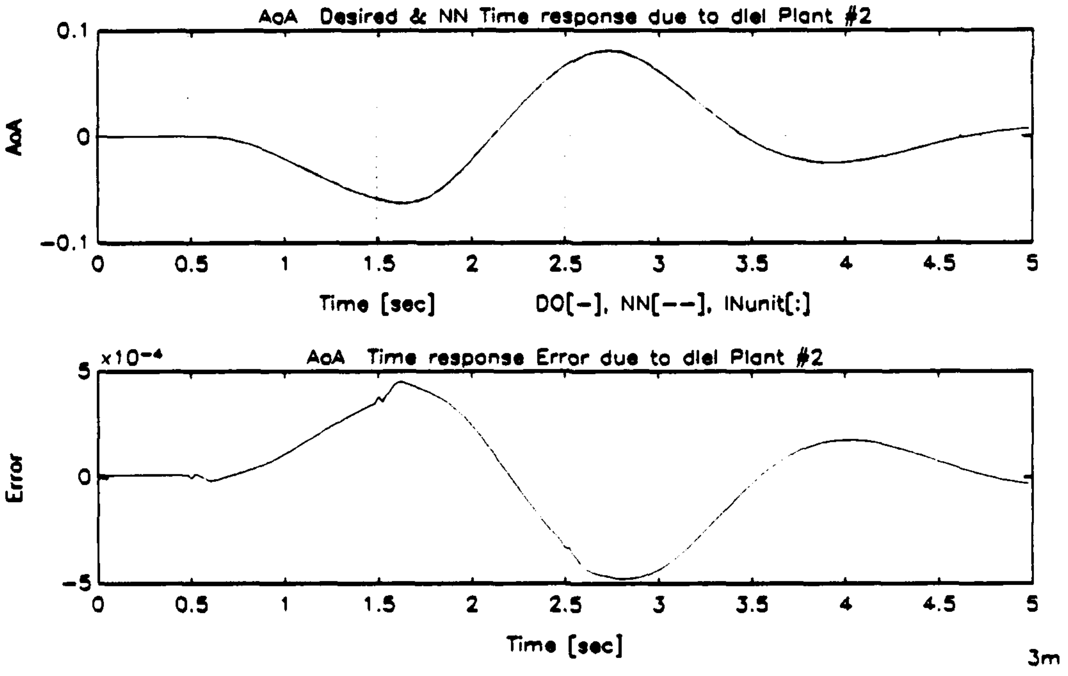


Figure B.61: **Angle of Attack Response of Damaged Aircraft Due to Leading Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

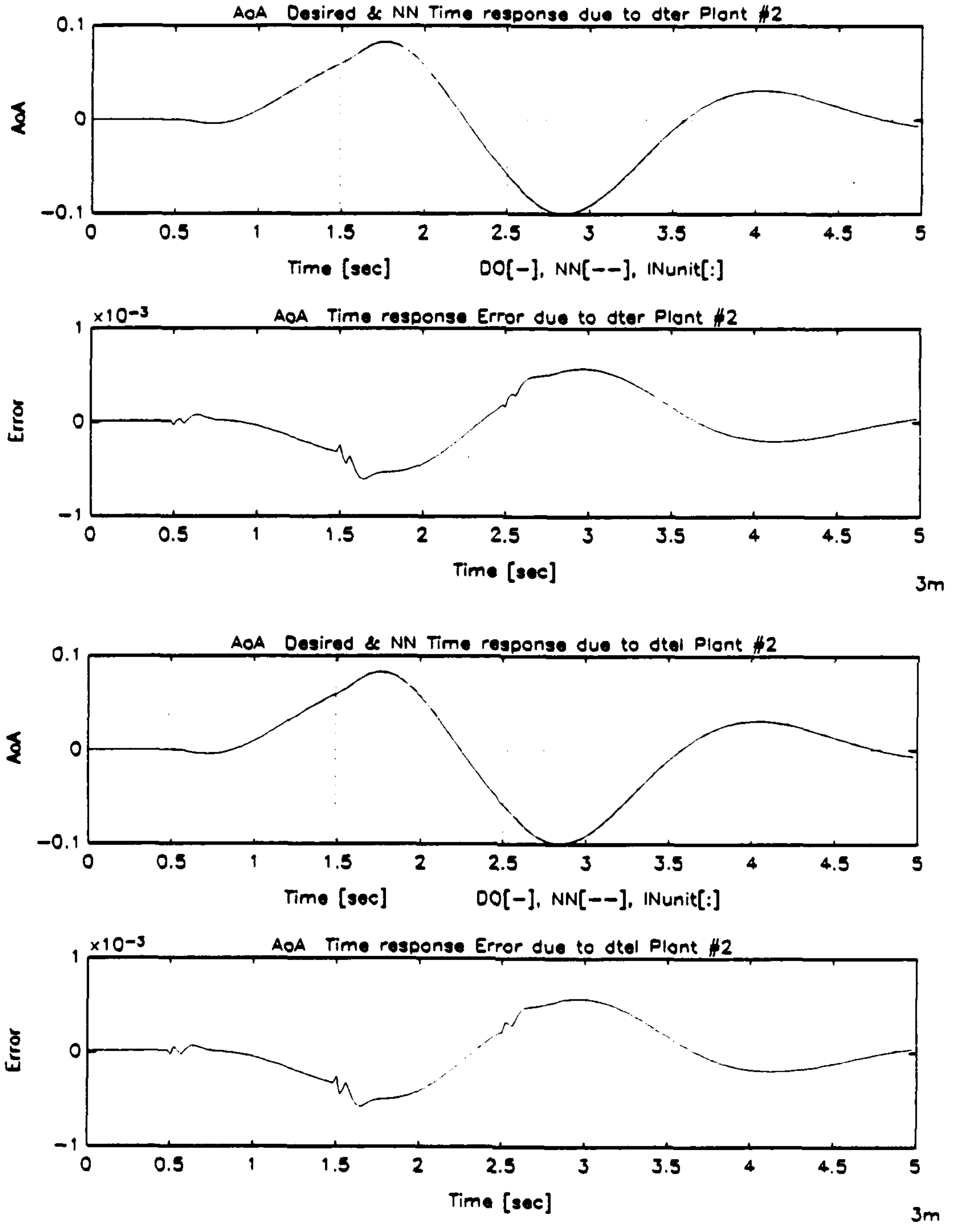


Figure B.62: **Angle of Attack Response of Damaged Aircraft Due to Trailing Edge Flaps Dipole Command, Multiple-Plant, Minimized Network Emulation**

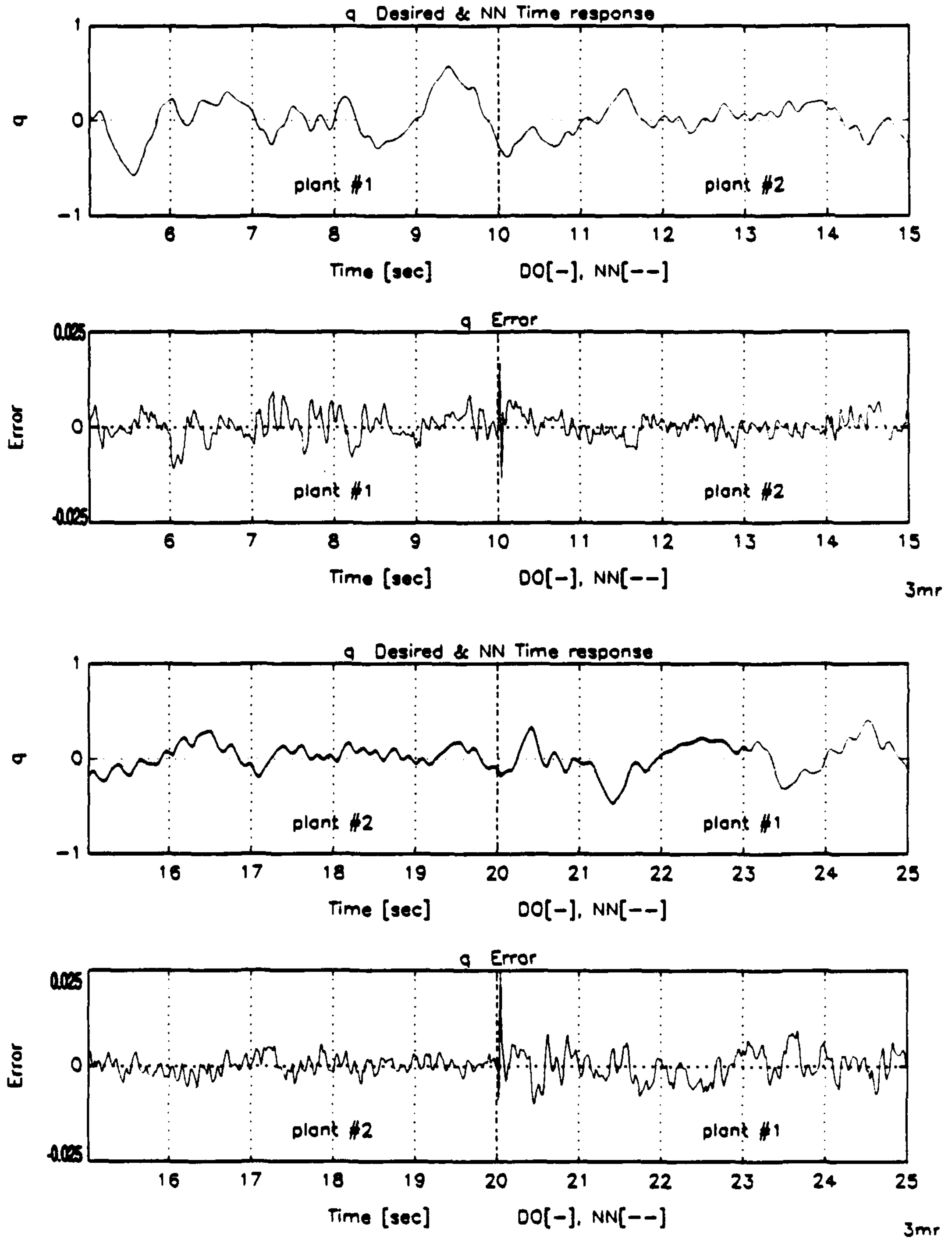


Figure B.63: **Pitch Rate Response to RBS Excitation of All Inputs, Multiple-Plant, Minimized Network Emulation**

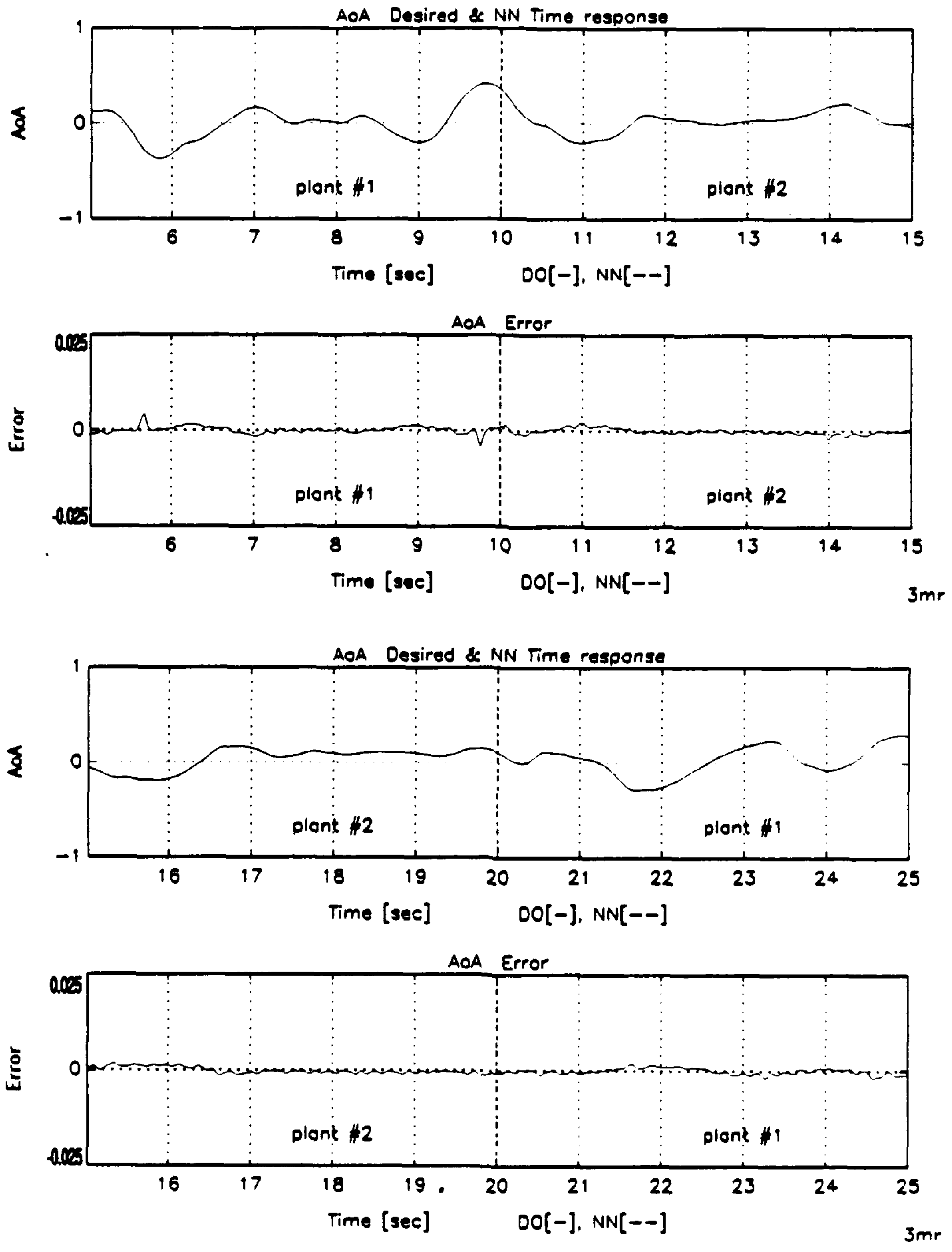


Figure B.64: **Angle of Attack Response to RBS Excitation of All Inputs, Multiple-Plant, Minimized Network Emulation**

APPENDIX C. SINGULAR VALUES PLOTS

The graphs in Appendix C are referenced in chapter VIII. The reader is referred to chapter VIII for comprehensive explanation and related discussion regarding the plots that appear in this appendix.

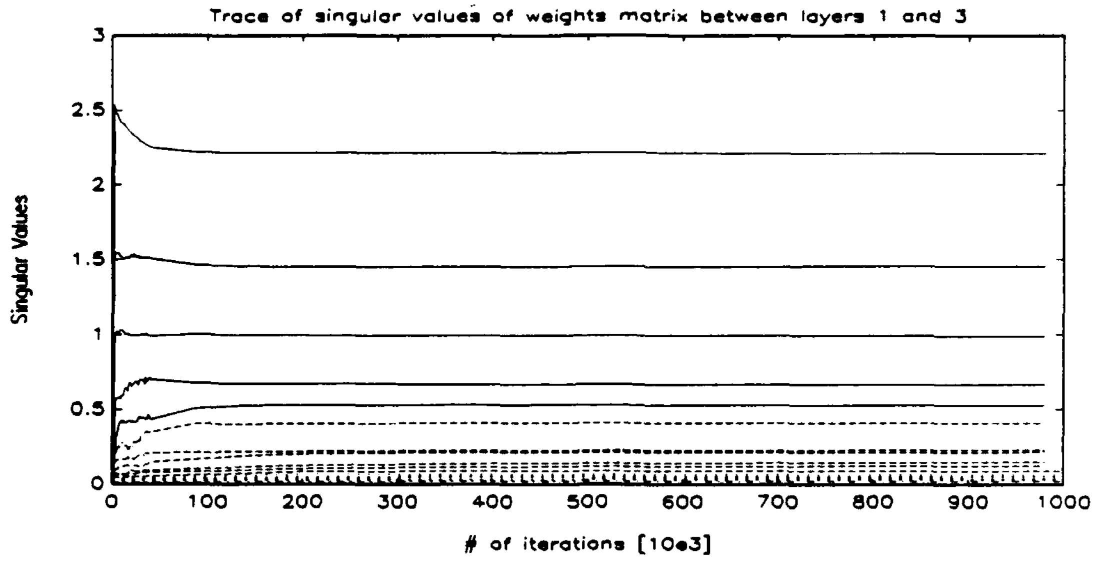


Figure C.1: **Singular Values Plot of Weight Matrix from *ComReg* to *comh1*, Before Minimization**

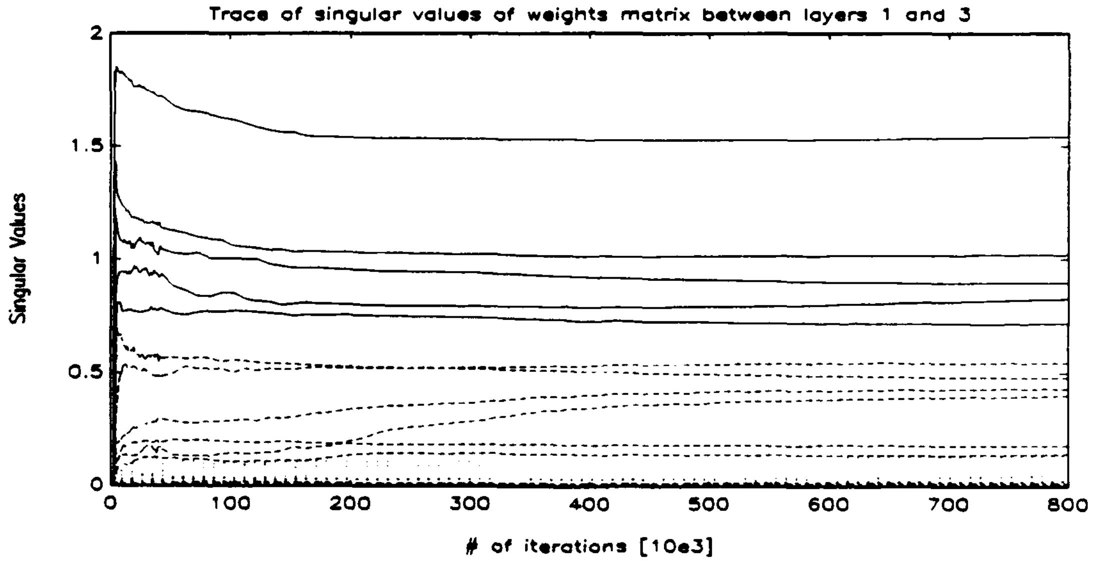


Figure C.2: **Singular Values Plot of Weight Matrix from *ComReg* to *comh1*, After Minimization**

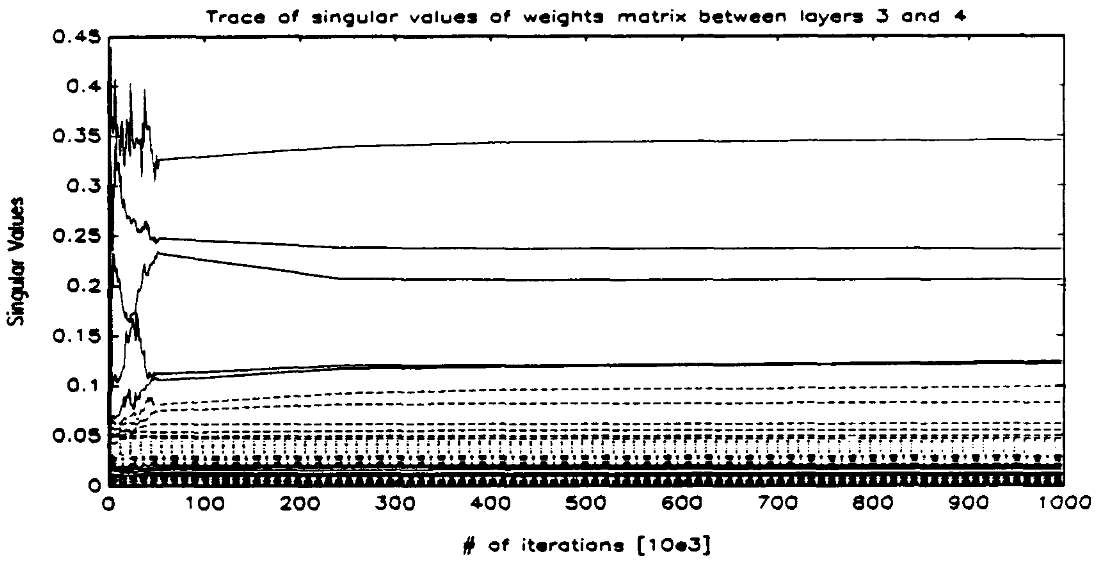


Figure C.3: Singular Values Plot of Weight Matrix from *comh1* to *comh2*, Before Minimization

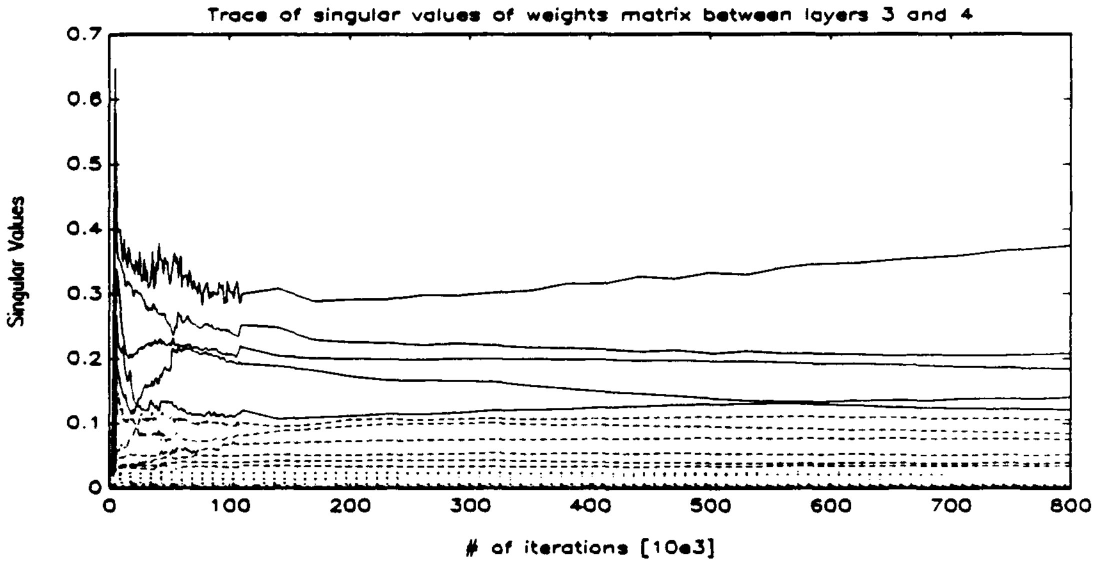


Figure C.4: Singular Values Plot of Weight Matrix from *comh1* to *comh2*, After Minimization

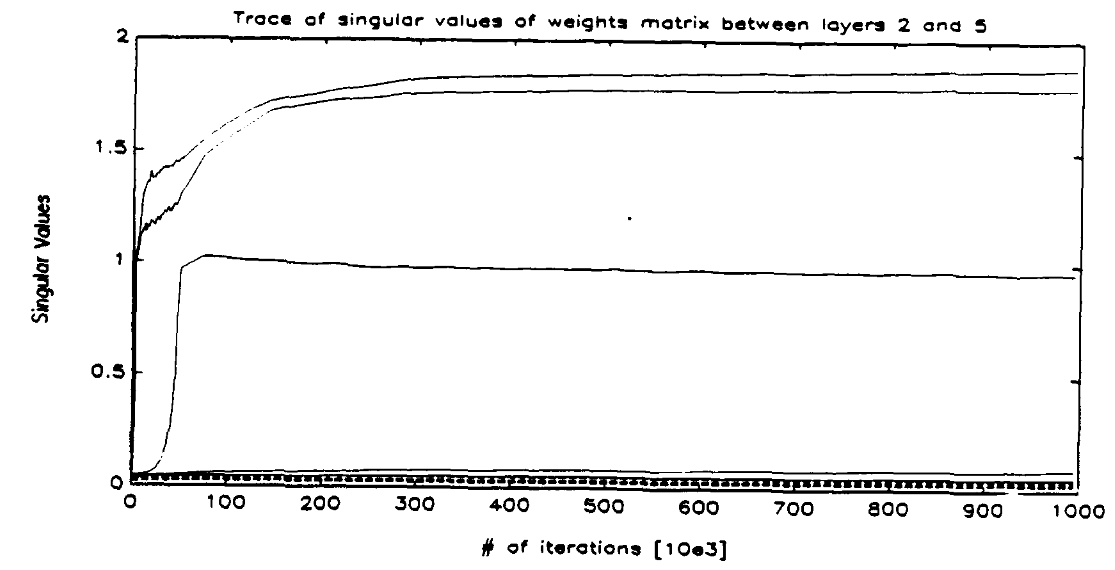


Figure C.5: **Singular Values Plot of Weight Matrix from *OutReg* to *outh1*, Before Minimization**

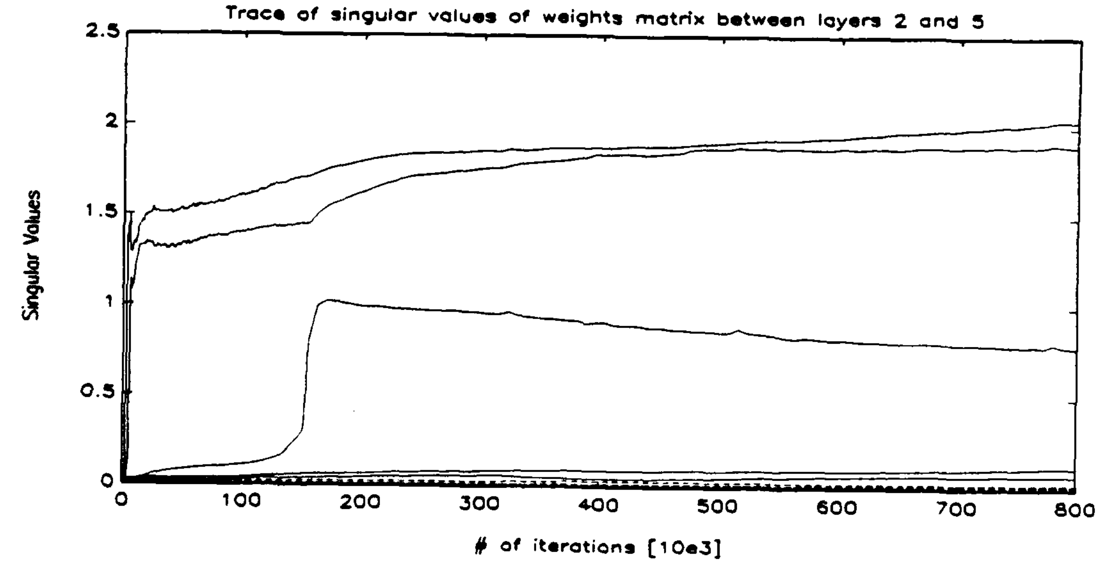


Figure C.6: **Singular Values Plot of Weight Matrix from *OutReg* to *outh1*, After Minimization**

APPENDIX D. USERIO EMULATION PROGRAMS

The programs in this appendix were written in *C* to interface with the main software - NeuralWorks Professional II/PLUS™ by NeuralWare, Inc. that used in this research. There are seven programs involved here. The main program *emul.c* is the USERIO program and is based on the general network structure described in Chapter VI. There are five dedicated header-files included.

The first is *matlab_link.h* which makes the interface with MATLAB™ as a server program to calculate the singular values of the weight matrices while the main program is running. The second header file is task depended. The task shown here is emulation. The F/A-18A undamaged and damaged plant that were emulated are simulated in a user defined header-file *f18l_modelN.h*. This file may be edited for different plants with the restriction described in the program listing.

The third header-file is *net_paramN.h* includes all the user accessed parameters that influence the network structure and task dependent calculation. The forth header-file *net_util.h* defines parameters and functions necessary for the main program. The last file, *window_dat.h* defines parameter and user graphic routines. All the parameters and variables are explained in the body of the program mostly with self-explanatory names.

The control strategy program *emulbpt.nnc* controls the flow of the main USERIO program, and applies the back-propagation algorithm. The use of *emul.c* must be in conjunction with this control strategy.

```
/**********************************************************************/
/*    $Author   :  $ Shahar Dror                                       */
/*    $Date     :  $ 22 Sep 1991                                       */
/*    $Source   :  $ emul.c                                            */
/*    $Revision :  $ 16 Jun 1991                                       */
/*     The NN is trained to produce the next plant state based on     */
/*     the current and past input commands and real-plant outputs     */
/*    Whenever adding/changing Tasks, search for 'Task' to    */
/*    accommodate/verify correct interpretation! Search also all  */
/*    #include files as well!!                                  */
/**********************************************************************/

#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <string.h>
#include <malloc.h>

 /* Definitions */
/* ============= */
#define UIO_SERVER              1
#define SERVER_EMULATOR_FILE    1
#define SKIP_COMPAT_DATA        1
#define BEEP (printf("^G"),fflush(stdout))    /* Sound a Beep */
#define SET          1
#define RESET        0
#define LEARN        0      /* LEARN mode indicator */
#define TEST         1      /* TEST mode indicator */
#define IClearn      2      /* Initial conditions in LEARN mode indicator */
#define ICtest       3      /* Initial conditions in TEST mode indicator */
#define TEST1        4      /* TEST mode, input type is RBS indicator */
#define JACOB        5      /* Jacobian calculation identifier */
#define In_lay       0      /* Input layer */
#define ComReg_lay 1        /* Command(input) regression layer */
#define OutReg_lay 2        /* Network output regression layer */
#define classification     5       /* Define task mode */
#define emulation          1       /* Define task mode */
#define combined_cl_em     2       /* Define task mode */
#define controlling        3       /* Define task mode */
#define cntrlThruPlant     4       /* Define task mode */
/*#define     Task   classification            /* Define operating task mode */
#define       Task   emulation      /* Define operating task mode */


#include "userutl.h"
#include "uio_strc.h"
#include "uio_tran.h"
#include "uio_reqs.pro"
#include "uio_cmds.pro"
#include "matlab_link.h"
#if Task==emulation           /* Set up emulation mode */
#include "f18l_modelN.h"
#endif
#if Task==controlling         /* Set up controlling mode */
#include "ref_modelN.h"
#endif
#include "net_paramN.h"
#include "net_util.h"
#include "window_dat.h"

#define MAXRAND (0x4ffffff1)
#define Out_lay          (nlayp-1) /* Output layer number */
#define OutBuf_lay  (Out_lay-1)    /* Output buffer layer number */

 /* Declarations */
/* ============== */
extern long random();       /* Random number generator */
int Lcycle=0,Rcycle=0;      /* Learning, Recall/Testing cycles counter, */
int LR_flag;          /* Learn(1)/Recall(0) flag */
char *sp;             /* String pointers */
time_t *tloc;         /* Pointer to current time for random # initialization */
/*
```

```
/*************************************************************************/
/* Functions necessary for handling the User I/O package itself.    */
/*************************************************************************/

NINT UIO_Init(file)
TEXT *file;
{
        NINT ret_val = UIO_OK;
int mnr;        /* Model number index */
long ctrp[8];          /* An array of 8 to interact with control-strategy*/

srandom(time(tloc)); /* Random # initialization, using a new seed each start */
/* Initialize counters which interact with control strategy */
ug_ctrio(ctrp,0);ctrp[7]=ctrp[6]=ctrp[4]=0;ug_ctrio(ctrp,1);/*aux2=wrk1=wrk2=0*/

        if(check_net()==UIO_ERROR) return(UIO_ERROR);   /* Network validation */
        if(start_matlab()==UIO_ERROR) return(UIO_ERROR); /* Linking to matlab */
        for(mnr=1;mnr<=Nmodels;mnr++){                  /*Loading models matrices*/
                if(load_model(mnr)==UIO_ERROR) return(UIO_ERROR);
        }
        /* Load system model if exists */
        if(effector_lay_length>0){
                for(mnr=1;mnr<=Nref_models;mnr++){
                        if(load_ref_model(mnr)==UIO_ERROR)  return(UIO_ERROR);
                }
        }
set_net();    /* Set and initialize network and model related parameters */
ret_val=UIO_Attention();   /* Establishing working format */
        if(ret_val==UIO_ERROR) PutStr("Exiting...  Restart the run.");

return(ret_val);
}


NINT UIO_Term(process)
NINT process;
{
close_files();       /* Close all opened files */
stop_matlab();       /* Unlinking matlab and closing relevant files and pipes */
        if(LR_flag==LEARN) PutFmt("\nTraining cycles: %i",Lcycle);
        if(LR_flag==TEST ) PutFmt("\nTest cycles: %i",Rcycle);
PutStr( "   bye bye\n" );
}


NINT UIO_Attention()
{
        NINT ret_val = UIO_OK;
int tmin,tmax;       /* Min & max time levels for H->L swept square wave */
int last_LR;         /* The value of LR_flag when entering UIO_Attention. */

intrrpt_flag=SET;    /* Setting interrupt flag. Causes network checking */
close_files();       /* If files were open, close them */
/* Inquire mode type LEARN/TEST */
last_LR=LR_flag;
LR_again:
PutFmt("\nIs this run a LEARN(0) or TEST(1) <#!=0,1 - quit> ?  [%i] ",LR_flag);
sp=GetStr();
        if(*sp) sscanf(sp,"%i",&LR_flag);
        if(!(LR_flag==0 || LR_flag==1)){
        PutStr("\nLR_flag must be 0 or 1. Do you want to quit? [y]/n ");
        sp=GetStr();
        LR_flag=last_LR;
                if(!*sp || *sp=='y')            return(UIO_ERROR);
                else   goto LR_again;
        }

model_nbr=1; /* start training with unimpaired (first) model   */
rand_models[0]=0;
        if(Nmodels>1){
        model_nbr=0;
        Again2:
        PutFmt("\nEnter Model # to train/test (1-%i), 0-for all [%i] ",
        Nmodels,model_nbr);
```

```
          sp=GetStr();
                if(*sp) sscanf( sp, "%d", &model_nbr);
                if( model_nbr >Nmodels || model_nbr<0 ){
                PutFmt("  %s",input_name[0] );
                goto Again2;
                }
          A1model_flag=((model_nbr!=0)?SET:RESET);
          }
    /*
    if(LR_flag==TEST){              /* TEST mode */

/*=========\/===start========TEST==========\/=========TEST=====start=====\/===*/

        /* User select input to be used */
        Input_Type_again:
        PutFmt("\nEnter Input Type 1) %s, 2) %s, 3) %s",
        input_name[1],input_name[2],input_name[3]);
        PutFmt("\n4) %s, 5) %s, 6) %s [%i] ",
        input_name[4],input_name[5],input_name[6],input);
        sp=GetStr();
                if(*sp) sscanf( sp, "%d", &input);
                if( input >Ninput_types || input<1 ){
                PutFmt("  %s",input_name[0] );
                goto Input_Type_again;
                }
        PutFmt("\nInput:  %s  selected",input_name[input]);/*Display selection*/

    netSave_flag=RESET;   /* Resetting best network saving flag */
    Rcycle=0;                       /* Initializing test cycles counter */
    min_RMS=1.0; max_RMS=0.0;       /* Initializing rel_err extrimums */

        if(input==1){               /* Random Binary Sequence */
                if( Nmodels>1 && model_nbr==0){
                        if(Nmodels>2) rand_array(Nmodels,rand_models);
                        else{
                        rand_models[0]=2; rand_models[1]=1;
                        }
                PutFmt("\nEnter max random cycles to switch models [%i] ",rmRC);
                sp=GetStr();
                if(*sp) sscanf(sp,"%i",&rmRC);
                mRC=rmRC;
                }
                else mRC=Large_number;/* Select a large # */
        inTest=0;           /* Test on all inputs on all inputs */
        }
    if(model_nbr==0) model_nbr=1;/* Start with original model (for test) */

        if(input==2 || input==3 || input==4 || input==5){
        /*Sequential Random Binary, Swept square/sine wave inputs */
        PutFmt("\nEnter upper  TEST frequency  [%f Hz] ",Fmax);        sp=GetStr();
                if(*sp) sscanf(sp,"%lf",&Fmax);
                if(Fmax> 0.5/ts)
                PutFmt("\nUpper freq exceeded. Fmax set to %f Hz",Fmax=0.5/ts);
        PutFmt("\nEnter lowest TEST frequency  [%f Hz] ",Fmin);        sp=GetStr();
                if(*sp) sscanf(sp,"%lf",&Fmin);
        mRC=(int)(exp2(ceil(-log2(ts*Fmin))));
        PutFmt("\nSet Test cycles >= %li",
                        mRC*forced_excitation_length*(Nmodels!=0?Nmodels:1));
                if(input==3){
                tmin=floor(0.5/Fmax/ts);
                tmax=ceil(0.5/Fmin/ts);
                        if(tmax>mRC*0.15) tmax=ceil(mRC*0.15);
                        if(sweep_HL(tmin,tmax,mRC,sweep_change_times)==UIO_ERROR)
                        return(UIO_ERROR);
                sweep_count=1;Nsweep_change=0;   /* initializing sweep counters*/
                }
                if(input==4 || input==5){
                dmin=log10(Fmin);
                dflog=(log10(Fmax)-dmin)/(mRC-1);
                dflin=(Fmax-Fmin)/(mRC-1);
                }
        inTest=0;
        }

        if(input==6){       /* One second dipole input */
        mRC=(int)(dpl[3]/ts);
        PutFmt("\nSet Test cycles >= %li",
                        mRC*forced_excitation_length*(Nmodels!=0?Nmodels:1));
        inTest=0;
        }
    PutStr("\nDo you want to save TEST results? [y]/n ");  sp=GetStr();
        if(!*sp || *sp=='y'){       /* open files for test results */
                if(open_TEST_files()==UIO_ERROR) return(UIO_ERROR);
                if(InPut && NNo && DESo)  TestDataSave_flag=SET;
            }
            else   TestDataSave_flag=RESET;         /* Don't save TEST results */
        }
/*=========/\====end=========TEST==========/\=========TEST======end======/\===*/
/*
```

```
        if(LR_flag==LEARN){ /* LEARN mode */
/*=========\/===start========LEARN=========\/========LEARN=====start=====\/===*/

        Lcycle=0;
        input=1;      /* Training with Random Binary Sequence */
        PutFmt("\nInput:  %s  selected",input_name[input]);/*Display selection*/
              if( Nmodels>1 && model_nbr==0){
                    if(Nmodels>2) rand_array(Nmodels,rand_models);
                    else{
                    rand_models[0]=2; rand_models[1]=1;
                    }
              Switch_again:
               PutFmt("\nEnter min random cycles to switch models [%i] ",minSW);
              sp=GetStr();
                    if(*sp) sscanf(sp,"%i",&minSW);
              maxSW-=1;
               PutFmt("\nEnter max random cycles to switch models [%i] ",maxSW);
              sp=GetStr();
                    if(*sp) sscanf(sp,"%i",&maxSW);
              maxSW+=1;     /* In order to reach maxSW value */
                    if((maxSW-minSW)<=0){
                    PutFmt("  %s",input_name[0] );
                    goto Switch_again;
                    }
              mRC=minSW+random()%(maxSW-minSW);
              model_nbr=1;
              }
              else{
              mRC=maxSW=Large_number-2;minSW=maxSW-1;
              }
        PutStr("\nDo you want to use Reset-Training mode ? [y]/n ");
        sp=GetStr();
              if(!*sp || *sp=='y'){
              ResetTrain_flag=SET;
              PutFmt("\nEnter Reset-rate (in learn-cycles) per model [%i]: ",
              ResetTC);    sp=GetStr();
                    if(*sp) sscanf(sp,"%i",&ResetTC);
              nRTC=ResetTC*(A1model_flag==SET?1:Nmodels);
              }
              else   ResetTrain_flag=RESET;
        PutStr("\nDo you want to Reset if output exceed limits ? [y]/n ");
        sp=GetStr();
              if(!*sp || *sp=='y')          ResetLimit_flag=SET;
              else   ResetLimit_flag=RESET;

              if(forced_excitation_length>1){
              Input_number_again:
              PutFmt("\nEnter Input # to train (1-%i), 0-for all [%i] "
                    ,forced_excitation_length,inTest);         sp=GetStr();
                    if(*sp) sscanf( sp, "%d", &inTest);
                    if( inTest >forced_excitation_length || inTest<0 ){
                    PutFmt("  %s",input_name[0] );
                    goto Input_number_again;
                    }
              }
        PutFmt("\nEnter file name to save SVD [%s] ",svdn);
        sp=GetStr();
              if(*sp) sscanf(sp,"%s",svdn);
        fputs("clear \n",matlab);  /* Clearing matlab working space */
        netSave_flag=RESET;         /* Initially don't save best network */
        PutStr("\nDo you want to save best network ? [y]/n ");  sp=GetStr();
              if(!*sp || *sp=='y'){
              netSave_flag=SET;
              PutFmt("\nEnter convergence threshold [%lf]: ",convergence);
              sp=GetStr();
                    if(*sp) sscanf(sp,"%lf",&convergence);
              }
        min_sum=convergence*NErrConv;          /* Initialize best-net saving */
        Rstart_flag=RESET;  /* Didn't go through UIO_Rcall_start (yet). */
        }
/*=========/\====end=========LEARN=========/\========LEARN======end======/\===*/
PutStr("\n UIO_Attention finished. You may continue... ");
        return(ret_val);
}
```

```
/************************************************************************/
/*  Functions necessary for handling a learning session.          */
/************************************************************************/

NINT UIO_Learn_Start()
/* Learn start; occurs once at each start (also after intermediate interruption.
Prepares data for next coming cycle  */
{
      NINT ret_val = UIO_OK;
int hist,hist_in,hist_out;          /* input_data vector initialization counters */
int mnr;                       /* Model number counter */
int i,j,k;                     /* General indices */
long   ctrp[8];        /* An array of 8 to interact with control-strategy*/
error_flag=RESET;      /* Resetting error flag. No error yet! */

/* Check compatibility of initialization and session type (L/R) */
  if(LR_flag!=LEARN && Rstart_flag==RESET){   /* Wrong. exit! */
  PutStr("\n!! You run a LEARN session, but initialized for RECALL.");
  BEEP;       /* sound a warning Beep */
  PutStr("\nFor LEARN, run I/O...>Attention I/O. For RECALL, run Run..>Recall");
  PutStr(".  Press <RETURN>.");
  GetStr();   return(UIO_EXIT);
  }
        if(intrrpt_flag==SET){      /* Validate network compatibility */
                if(check_net()==UIO_ERROR) return(UIO_ERROR);
        intrrpt_flag=RESET;
        }
        if(LR_flag==LEARN){         /* Learning mode */
                if(Lcycle%mRC==0 || netReset_flag==SET){
                        for(mnr=1;mnr<=Nmodels;mnr++){             /* Load  I.C. */
                        desired_output(IClearn,mnr,excite,&desout[mnr-1][0]);
                        /* Check for errors in desired_output()  */
                                if(error_flag==SET) return(UIO_ERROR);
                                if(effector_lay_length>0){/* Initial Effector */
                                        for(i=0;i<effector_lay_length;i++)
                                        Effector[mnr-1][i]=0.0;
                                }
                        }
                }
        }

        if(LR_flag==TEST){  /* Testing mode */
                if(Rcycle%mRC==0 && input!=1){
                        if(inTest>=forced_excitation_length){
                           if((A1model_flag==SET || ++model_nbr>Nmodels) &&
                               UIO_Attention()==UIO_ERROR)   return(UIO_ERROR);
                           else inTest=0;
                        }
                /*Load test IC*/
                desired_output(ICtest,model_nbr,excite,&desout[model_nbr-1][0]);
                /* Check for errors in desired_output() ) */
                        if(error_flag==SET) return(UIO_ERROR);
                        if(effector_lay_length>0){         /* Initialize Effector*/
                                for(i=0;i<effector_lay_length;i++)
                                Effector[model_nbr-1][i]=0.0;
                        }
                        if(input==3) {             /* initializing sweep counters*/
                        sweep_count=1; Nsweep_change=0;
                        }
                        if(input==4 || input==5){
                        fflog=dmin;    /* Initializing (log10) frequency */
                        flin=Fmin;     /* Initializing linear frequency */
                        T=0;           /* Initializing Time */
                        }
                        if(input==6)  T=0;         /* Initializing Time */
                }
        }

/* Place the initial (measurements) conditions in the input_data vector */
        if((LR_flag==TEST && Rcycle%mRC==0) || (LR_flag==LEARN &&
        (Lcycle%mRC==0 || netReset_flag==SET))){

          for(mnr=((LR_flag==LEARN && A1model_flag==RESET)?1:model_nbr);
          mnr<=((LR_flag==LEARN&&A1model_flag==RESET)?Nmodels:model_nbr);mnr++){
```

```
         /* Set counters for cases in which input_history!=output_history */
         if(in_history>out_history){
         hist=in_history;  hist_in=0; hist_out=out_history-in_history;
         }
         else{
         hist=out_history; hist_out=0; hist_in=in_history-out_history;
         }

         for(j=0;j<=hist;j++){  /* Fill in input and regression layers*/
/* Last pass @ j=hist is preparing for first presentation, not loaded to REG
U(0) and Y(0) (initial values/conditions) are lost */

      /* Create excitations */
           for(++hist_in,i=0;i<forced_excitation_length;i++){
                if(input==1 && (j%RbsUd)==0)/* Random Binary Sequence */
                excite[i]=(inTest==0?RBS:(inTest==(i+1)?RBS:0.0));
                if(input==2 && (j%RbsUd)==0)
                excite[i]=(inTest==i?RBS:0.0); /*Seq RBS*/
                if(input==3) excite[i]=(inTest==i?(one=1.0):0.0);/*HL*/
                if(input==4) excite[i]=(inTest==i?1.0:0.0);/*SwL-HSqr*/
                if(input==5) excite[i]=0.0;       /*Swept L-H Sine Wave */
                if(input==6) excite[i]=0.0;       /*Dipole */
                /* To induce real zero, subtract the bias. */
                if(excite[i]==0.0) excite[i]=-bias_I[mnr-1][i];
           input_data[mnr-1][i]=excite[i];
           }
           if(unforced_excitation_length>0){
            unforced_excitation(LR_flag,mnr,&input_data[mnr-1][0],
                      &input_data[mnr-1][forced_excitation_length]);
           /* Check for errors in unforced_excitation() ) */
                if(error_flag==SET) return(UIO_ERROR);
           }

           if(type_length>0){   /* Set to Active the model in tact */
           Type((LR_flag==LEARN?ICLearn:ICtest),mnr,&desout[mnr-1][0],
              &Effector[mnr-1][0],&input_data[mnr-1][excitation_length]);
           /* Check for errors in Type() ) */
                if(error_flag==SET) return(UIO_ERROR);
           }

      /* Now load into ComReg the relevant inputs */
           if(hist_in>0 && j<hist){
              for(i=0;i<regressed_input_length;i++){
              k=(hist_in-1)*regressed_input_length+i;
              ComReg[mnr-1][k]=input_data[mnr-1][RIsort[i]];
              }
           }
      /* Put previos output values int OutReg */
           if(++hist_out>0){
                for(i=0;i<regressed_output_length;i++){
                     if(j<hist){
                     k=i+(hist_out-1)*regressed_output_length;
                     OutReg[mnr-1][k]=desout[mnr-1][ROsort[i]];
                     }
                     else
                     desoutOld[mnr-1][i]=desout[mnr-1][ROsort[i]];
                }
           }
      /* Using current excitation, calculate desired_output at t=(current+1)*/
          desired_output(LR_flag,mnr,&input_data[mnr-1][0],&desout[mnr-1][0]);
          } /* (end for-j) Do until both regression vectors are filled */
        }    /* (end for-mnr) Do for all relevant models */
      ug_ctrio(ctrp,0);    /* read counters content */
      ctrp[6]=1;           /* Set wrk1=1 to load Regressors values */
      ug_ctrio(ctrp,1);    /* write counters content */
      }      /* (end if-%mRC==0) */
netReset_flag=RESET;       /* Do not reset network next time */
/* Display the starting condition */
      if(LR_flag==LEARN) PutFmt("\nTraining cycles: %i",Lcycle);
      if(LR_flag==TEST)  PutFmt("\nTesting cycles: %i",Rcycle);
PutFmt(", Input type: %s, model #: %i, input # in effect: %i",
      input_name[input],model_nbr,(input==1?inTest:(inTest+1)));
return(ret_val);
}
```

```
NINT UIO_Learn_Input(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
      NINT ret_val = UIO_OK;
int i;        /* Counter index */
/* WARNING: Layers that are not to be filled, are to be controlled via the
   control-strategy, and not by not filling them in UIO_Learn_Input, since
   this function will write the last data that was addressed in this function */

/* Fill the input layer with the required input data vector */
      if(LayN==In_lay){
            for(i=0;i<nPEs;Datap[i]=input_data[model_nbr-1][i], i++);
      }
/* Fill the Command Regression layer with the last output(model#) data vector */
      if(LayN==ComReg_lay && regressed_input_length>0){
            if(regressed_input_history_length<nPEs){
            PutFmt("\n ComReg vector too small. Vector length :%i, nPEs=%i",
            regressed_input_history_length,nPEs);
            PutStr("\nPress RETURN to stop and exit"); GetStr();
            return(UIO_ERROR);
            }
            for(i=0;i<nPEs;Datap[i]=ComReg[model_nbr-1][i], i++);
      }
/* Fill the Output Regression layer with the last output(model#) data vector */
      if(LayN==OutReg_lay && regressed_output_length>0){
            if(regressed_output_history_length<nPEs){
            PutFmt("\n OutReg vector too small. Vector length :%i, nPEs=%i",
            regressed_output_history_length,nPEs);
            PutStr("\nPress RETURN to stop and exit"); GetStr();
            return(UIO_ERROR);
            }
            for(i=0;i<nPEs;Datap[i]=OutReg[model_nbr-1][i], i++);
      }
/* Fill the Output Buffer layer with last desired output(model#) data vector */
      if(LayN==OutBuf_lay && regressed_output_length>0){
            for(i=0;i<nPEs;Datap[i]=desoutOld[model_nbr-1][i], i++);
      }
return(ret_val);
}

NINT UIO_Learn_Output(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
      NINT ret_val = UIO_OK;
int i;        /* Counter index */

/* Fill output layer with the desired output data (sensors measurements) */
      for(i=0;i<nPEs;i++)
      Datap[i]=desoutOld[model_nbr-1][i]=desout[model_nbr-1][i];
return(ret_val);
}
/*
```

```
NINT UIO_Learn_Result(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;       /*    Datap = pointer to array that contains output data */
{
      NINT ret_val = UIO_OK;
double rel_err,err_MS;            /* Relative output error */
double abs_msr;             /* Absolute value of the measurement */
static int cycl[Nmodels+1],input_index=0,model_index=0;        /* Cycles counter */
static int his_cycl[2][Nmodels+1];/* History cycle counter for Learn and Test */
long ctrp[8];        /* A pointer to array of 8 counters that control NW run */
int last_model;      /* In TEST, & input =1, saves previos model number */
int i,j;             /* Counter index */

double nn[output_lay_length];

/* This function reads the output of the current step,
       and prepares the input for the next step */

      if(error_flag==SET) return(UIO_ERROR);
      if(effector_lay_length>0 && LayN==effector_lay){/*Get control outputs*/
            if(nPEs!=effector_lay_length){  /* Check matching */
            PutStr("\nWrong control-layer or control-length mismatch");
            PutFmt(". LayN=%i, nPEs=%i, effector-length=%i",
                                          LayN,nPEs,effector_lay_length);
            efLay_flag==SET;     /* Enable setting Effector_lay # */
            check_net();         /* Resetting effector_lay # */
            PutStr("Rerun after termination. ");
            return(UIO_EXIT);
            }     /*end if(nPEs...*/
            for(i=0;i<nPEs;Effector[model_nbr-1][i]=Datap[i],i++);
            /* if effector layer is the output layer, continue */
            if(effector_lay != Out_lay)      return(ret_val);
      }     /*end if(effector...*/
/* Increment the counter and update the display as necessary */
      if(LR_flag==LEARN){ /* Display Learn cycles in LEARN mode */
            if(ResetLimit_flag==SET && LayN==Out_lay &&
                  Limits(model_nbr,Datap)==0){
            ug_ctrio(ctrp,0);    /* read counters content */
            ctrp[7]=1;           /* Set wrk2=1 to reset network */
            ug_ctrio(ctrp,1);    /* write counters content */
            PutStr("\nOutput Limit Exceeded. Resetting...");
            netReset_flag=SET;
            return(UIO_Learn_Start());
            }
            if(LayN==ComReg_lay){           /* Save ComReg(model#) output for next*/
                  if(ResetTrain_flag==SET && Lcycle%nRTC==0){
                  netReset_flag=SET;
                  return(UIO_Learn_Start());        /* Reset plant/ref */
                  }
                  for(i=0;i<nPEs;ComReg[model_nbr-1][i]=Datap[i],i++);
            return(ret_val);
            }
            /* Save OutReg(model#) output for next*/
            if(LayN==OutReg_lay){
                  if(ResetTrain_flag==SET && Lcycle%nRTC==0)
                                                      return(ret_val);
                  if(regressed_output_length>0)
                      for(i=0;i<nPEs;OutReg[model_nbr-1][i]=Datap[i],i++);
            return(ret_val);
            }
      Lcycle++;
            if((Lcycle%500)<1) PutFmt("\nTraining cycles: %i",Lcycle);
      }
      if(LR_flag==TEST){
            if(TestDataSave_flag==SET){ /* Save data for analyzing results*/
                  /*Save input commands in TEST mode*/
                  if(input==1)  /* Store all inputs */
                      for(i=0;i<forced_excitation_length;i++)
                      fprintf(InPut,"% 1.5e ",input_data[model_nbr-1][i]);
                  else
                   fprintf(InPut,"% 1.5e ",input_data[model_nbr-1][inTest]);
            fprintf(InPut,"\n");
```

```
                /*Save desired-outputs in TEST mode*/
                        for(i=0;i<noutp;i++)
                                fprintf(DESo,"% 1.5e ",desout[model_nbr-1][i]);
                fprintf(DESo,"\n");
                /* Save the NN outputs */
                        for(i=0;i<nPEs;fprintf(NNo,"% 1.5e ",Datap[i]),i++);
                fprintf(NNo,"\n");
                }
        ug_ctrio(ctrp,0);   /* read couters content */
        ctrp[6]=0;          /* Set wrk1=0. Do not load Regressors values */
        ug_ctrio(ctrp,1);   /* write couters content */
        Rcycle++;
                if(Rcycle%mRC==0) {
                        if(input==1 && Nmodels>1){
                        last_model=model_nbr;
                        model_nbr=rand_models[model_index++];
                        PutFmt("\nTesting cycles: %i",Rcycle);
                        PutFmt(", model #: %i, input # in effect: %i",
                        model_nbr,(input==1?inTest:(inTest+1)));
                        /* Prepare values for OutBuf layer
                               (for test with desired values!) */
                               for(i=0;i<output_lay_length;i++)
                               desoutOld[model_nbr-1][i]=desout[last_model-1][i];
                           if((model_index=model_index%Nmodels)==0 && Nmodels>2)
                           rand_array(Nmodels,rand_models);/*new rand modelseq*/
                        }
                        else{
                        inTest++;
                        return(UIO_Learn_Start());
                        }
                }
                if((Rcycle%500)<1) PutFmt("\nTest cycles: %i",Rcycle);
        }
cycl[model_nbr]++;  /* Increment excitation update cycle counter */
/* Generate new control-input and place it in the first elements of input_data*/
        if(input==1){       /* Random binary sequence */
                if((cycl[model_nbr]=cycl[model_nbr]%RbsUd)==0){
                        for(i=0;i<forced_excitation_length;i++){
                        excite[i]=(inTest==0?RBS:(inTest==i+1?RBS:0.0));
                        /* To induce real zero, subtract the bias. */
                           if(excite[i]==0.0) excite[i]=-bias_I[model_nbr-1][i];
                        input_data[model_nbr-1][i]=excite[i];
                        }
                }
        }
        if(input==2 && (cycl[model_nbr]=cycl[model_nbr]%RbsUd)==0)
                        excite[inTest]=RBS;
        if(input==3) excite[inTest]=SweptSquareWaveH2L();
        if(input==4) excite[inTest]=SweptSquareWaveL2H(Rcycle);
        if(input==5) excite[inTest]=SweptSineWaveL2H(Rcycle);
        if(input==6){
                T+=ts;
                excite[inTest]=DIPOLE((T)); /* 1 second dipole */
                }

        if(input!=1){       /* Update input data with the new control */
        /* To induce real zero, subtract the bias. */
           if(excite[inTest]==0.0) excite[inTest]=-bias_I[model_nbr-1][inTest];
        input_data[model_nbr-1][inTest]=excite[inTest];
        }
        /* If input consist of unforced excitation, calculate it.         */
        if(unforced_excitation_length>0)
        unforced_excitation(LR_flag,model_nbr,&input_data[model_nbr-[illegible],
                      &input_data[model_nbr-1][forced_excitation_length]);
        /* If input consist of type indication, calculate it.  */
        if(type_length>0){  /* Set to Active the model in tact */
                for(i=0;i<nPEs;nn[i]=Datap[i],i++);/*convert float to double */
        Type(LR_flag,model_nbr,nn,&Effector[model_nbr-1][0],
                              &input_data[model_nbr-1][excitation_length]);
        }
/* Save best network if desired (convergence test) */
        if(((netSave_flag==SET || (Nmodels>1 && rand_models[0]!=0))&&inTest==0)
         && LR_flag==LEARN && Lcycle>NErrConv){
                for(i=0,err_MS=0.0;i<noutp;i++){
```

```
                rel_err=(desout[model_nbr-1][i]-(double)Datap[i]);
                err_MS+=rel_err*rel_err;
                }
                if((err_RMS=sqrt(err_MS))<=convergence){
                        if(++conv_test[0]==Lcycle){
                                for(i=NErrConv;i>1;sum[i]=sum[i-1],i--);
                        sum[0]+=(sum[1]=err_RMS);
                            if(++conv_test[1]>=NErrConv){
                                if(sum[0]<min_sum && netSave_flag==SET){
                                SaveNet(BestNetName,NetSaveMode);
                                min_sum=sum[0];
                                PutFmt("\nTraining cycles: %i",Lcycle);
                                PutFmt(",  Input type: %s,  model #: %i",
                                                        input_name[input],model_nbr);
                                printf("\nTraining cycles: %i",Lcycle);
                                printf(",  model #: %i, mean_min_RMS=%f",
                                                        model_nbr,min_sum/NErrConv);
                                fflush(stdout);
                                }
                            sum[0]-=sum[NErrConv];
                            }
                        }
                        else{
                        conv_test[0]=Lcycle;
                        conv_test[1]=1;
                        sum[0]=err_RMS;
                        }
                }
        }
/* Prepare output for next presentation of that model */
desired_output((input==1&&LR_flag==TEST)?TEST1:LEARN,model_nbr,
&input_data[model_nbr-1][0],&desout[model_nbr-1][0]);
/* Switch models if it is the time to do so. */
        if(input==1 && LR_flag==LEARN && Lcycle%mRC==0 && Nmodels>1){
        model_nbr=rand_models[model_index];
                if((model_index=((++model_index)%Nmodels))==0 && Nmodels>2)
                rand_array(Nmodels,rand_models);/*new random model-seq*/
        mRC=Lcycle+minSW+random()%(maxSW-minSW);
        }
return(ret_val);
}


NINT UIO_Learn_End()
{
        NINT ret_val = UIO_OK;
        if(LR_flag==LEARN){
        /* Save matlab workspace containing SVDs matrices */
        fputs("save ",matlab);fputs( svdn ,matlab);fputs("\n",matlab);/* save */
        PutFmt("\nTraining cycles: %i",Lcycle);
        }
        if(LR_flag==TEST) PutFmt("\nTest cycles: %i",Rcycle);
intrrpt_flag=SET;    /* An interrupt occurred. Check network. */
PutStr(".  To terminate press <RETURN> ");
GetStr();PutStr("\b ... Done! ");
return(ret_val);
}
/*
```

```
*/
/************************************************************************/
/*  Functions necessary for handling a recall or testing session.   */
/************************************************************************/

NINT UIO_Recall_Start()
{
        NINT ret_val = UIO_UPDATE;
/* Check compatibility of initialization and session type (L/R) */
  if(LR_flag!=TEST){       /* Wrong. Exit! */
  PutStr("\n!! You run a RECALL session, but initialized for LEARN.");
  BEEP;       /* sound a warning Beep */
  PutStr("\nFor RECALL, run I/O...>Attention I/O. For LEARN, run Run...>Learn");
  PutStr(".  Press <RETURN>.");
  GetStr();  return(UIO_EXIT);
  }
Rstart_flag=SET;
        return(UIO_Learn_Start());
}


NINT UIO_Read(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
        /* In TEST mode load desired outputs */
        if(LayN==Out_lay)   return(UIO_Learn_Output(LayN, nPEs, Datap));
        /* In TEST/Recall mode get last values */
        else                return( UIO_Learn_Input(LayN, nPEs, Datap));
}


NINT UIO_Write(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
        NINT ret_val = UIO_OK;
        return( UIO_Learn_Result(LayN, nPEs, Datap));
}


NINT UIO_Write_Step(LayN, nPEs, Datap)       /* Not used in BackPropagation */
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
        NINT ret_val = UIO_OK;
        return(ret_val);
}


NINT UIO_Recall_Test(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
         return(UIO_Learn_Output(LayN, nPEs, Datap));
}

NINT UIO_Recall_End()
{
        NINT ret_val = UIO_OK;
Rstart_flag=RESET;
        return(UIO_Learn_End());
}
/*
```

```
/************************************************************************/
/*  Other miscellaneous functions.                                      */
/************************************ ************************************/

NINT UIO_Instrument(Instrument_id, nDataElems, DataElemp)
NINT  Instrument_id;
NINT  nDataElems;
SREAL *DataElemp;
{
NINT ret_val = UIO_OK;
double dDE[nInstElmnt];    /* an array to convert DataElemp from float to double */
char inst[maxInstName],c;  /* Variable of an instrument name */
int i;               /* Counter index */
int rows,cols;       /* # rows, # columns in weight matrix */
       if(nDataElems>nInstElmnt){ /* Check dDE is big enough */
       PutFmt(
       "\nInstrument transfers too many points. nInstElmnt=%i,but has to be:%i"
       ,nInstElmnt,nDataElems);
       printf(
       "\nInstrument transfers too many points. nInstElmnt=%i,but has to be:%i"
       ,nInstElmnt,nDataElems);
       error_flag=SET;     PutStr("\nPress RETURN to stop and exit"); sp=GetStr();
       return(UIO_ERROR);
       }
/* convert float Data to double */
       for(i=0;i<nDataElems;dDE[i]=DataElemp[i],i++);
rows=cols=0;
/*     From the instrument name, extract rows, cols and weight matrix name.
 *     The name of instrument is in the format:
 *     "rows(any non number character)colsWtname", i.e.  245.12W011
 *     => rows=245, cols=12, Wtname=W011                          */
sscanf(ug_getinstrt(),"%d%c%d%s",&rows,&c,&cols,inst);

       /* Check for consistency between instrument name and its data size */
       if(nDataElems!=(rows*cols)){
       PutFmt("Instrument nDataPoints/name error. cols=%i, rows=%i, nData=%i",
       rows,cols,nDataElems);
       printf("Instrument nDataPoints/name error. cols=%i, rows=%i, nData=%i",
       rows,cols,nDataElems);     BEEP;
       error_flag=SET; PutStr("\nPress RETURN to stop and exit"); sp=GetStr();
       return(UIO_EXIT);
       }
rewind(mat_i_f);    /* to prevent over filling mat_i_f file */
/* Here is essentially a small matalb code that calculates and saves the SVDs of
 * the weight matrices. Assuming the name of the weight matrix is WtName, the
 * code is:
 * >> save matif WtName ;
 * >> load matif ;
 * >> sWtName=[sWtName svd(WtName)];
 * >> clear WtName ;       % this is not a must. It is for Wt matrices will not
 *              appear when saving the matlab workspace at UIO_Learn_End().
 */
savemat(mat_i_f,1000,inst,rows,cols,0,dDE, (double *)0);          /* save */
fputs("load ", matlab);fputs(MATLABIF, matlab);fputs(" \n", matlab);  /* load */
fputs("s",matlab);fputs(inst,matlab);fputs("=[s",matlab);
fputs(inst,matlab);fputs(" svd(",matlab);fputs(inst,matlab);
fputs(")];\n",matlab);                                   /* s..=[s.. svd(Wt)]; */
fputs("clear ", matlab);fputs(inst, matlab);fputs(" \n", matlab);    /* clear */
       return(ret_val);
}

NINT UIO_Rewind()
{
       NINT ret_val = UIO_OK;
       return(ret_val);
}
NINT UIO_Explain(LayN, nPEs, Datap)
NINT  LayN;
NINT  nPEs;
SREAL *Datap;
{
       NINT ret_val = UIO_OK;
       return(ret_val);
}
```

```
/*************************************************************************/
/*    $Author   :  $ Shahar Dror                                         */
/*    $Date     :  $ 22 Sep 1991                                         */
/*    $Source   :  $ matlab_link.h                                       */
/*************************************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

#define MATLABPIPE  "matlab.pipe"

FILE *matlab;
FILE *matpipe;
FILE *mat_i_f;      /* matlab interface file */
char *MATLABIF="matlabIF"; /* location of interface file for matlab*/
char svdn[20]="S";  /* file name of SVD results*/

int start_matlab()
{
time_t *tlc; /* pointer to current time in sec from GMT00:00:00 '70*/
char tstr[5];       /* pointer to hold integer to string conversion */
int fd;
int flags;
        if((matlab = popen("matlab >/dev/null 2>&1", "w"))==NULL){
                printf ("error invoking matlab\n");exit(1);
        }
        setbuf(matlab, (char *) NULL);

/*     mknod(MATLABPIPE, S_IFIFO | S_IREAD | S_IWRITE);
        if((matpipe = fopen(MATLABPIPE, "r+"))==NULL){
             PutFmt("error opening %s",MATLABPIPE); return(UIO_ERROR);
        }
        setbuf(matpipe, (char *) NULL);       */

sprintf(tstr,"%i",(time(tlc)%100000));
strcat(MATLABIF,tstr);strcat(MATLABIF,".mat");        /* get a unique name */
       if((mat_i_f=fopen( MATLABIF,"w"))==NULL){
             PutFmt("error opening %s",MATLABIF); return(UIO_ERROR);
       }
        setbuf(mat_i_f, (char *) NULL);
return(UIO_OK);
}

void stop_matlab()
{
       fclose(mat_i_f);
       fputs("delete  ", matlab);fputs(MATLABIF, matlab); fputs("\n", matlab);
        fputs("quit\n", matlab);
        pclose(matlab);
        fclose(matpipe);
        unlink(MATLABPIPE);
}

typedef struct {
     long type;   /* type */
     long mrows;  /* row dimension */
     long ncols;  /* column dimension */
     long imagf;  /* flag indicating imag part */
     long namlen; /* name length (including NULL) */
} Fmatrix;

/* Here is an example that uses 'savemat' to save two matrices to disk,
 * the second of which is complex:
 *
 *     FILE *fp;
 *     double xyz[500], ar[500], ai[500];
 *     fp = fopen("foo.mat","wb");
 *     savemat(fp, 1000, "xyz", 2, 3, 0, xyz, (double *)0);
 *     savemat(fp, 1000, "a", 5, 5, 1, ar, ai);
 *      fclose(fp);
 * See .../matlab/loadsave/savemat.c  for more details.       */
void savemat(fp, type, pname, mrows, ncols, imagf, preal, pimag)
FILE *fp;       /* File pointer */
int type;       /* Type flag: 1000 for Sun */
```

```
char *pname;    /* pointer to matrix name */
int mrows;      /* row dimension */
int ncols;      /* column dimension */
int imagf;   /* imaginary flag */
double *preal;  /* pointer to real data */
double *pimag;  /* pointer to imag data */
{
        Fmatrix x;
        int mn;

        x.type = type;
        x.mrows = mrows;
        x.ncols = ncols;
        x.imagf = imagf;
        x.namlen = strlen(pname) + 1;
        mn = x.mrows * x.ncols;

        fwrite(&x, sizeof(Fmatrix), 1, fp);
        fwrite(pname, sizeof(char), (int)x.namlen, fp);
        fwrite(preal, sizeof(double), mn, fp);
        if (imagf) {
            fwrite(pimag, sizeof(double), mn, fp);
        }
}
/* Here is an example that uses 'loadmat' to load a matrix from a MAT-file:
 *
 *      FILE *fp;
 *      char name[20];
 *      int type, mrows, ncols, imagf;
 *      double *xr, *xi;
 *      fp = fopen("foo.mat","rb");
 *      loadmat(fp, &type, name, &mrows, &ncols, &imagf, &xr, &xi);
 *       fclose(fp);
 *      free(xr);
 *      if (imagf) free(xi);
 * See .../matlab/loadsave/loadmat.c  for more details.       */
int loadmat(fp, type, pname, mrows, ncols, imagf, preal, pimag)
FILE *fp;         /* File pointer */
int *type;        /* Type flag: see reference section of guide */
int *mrows;       /* row dimension */
int *ncols;       /* column dimension */
int *imagf;       /* imaginary flag */
char *pname;      /* pointer to matrix name */
double **preal;   /* pointer to real data */
double **pimag;   /* pointer to imag data */
{
        char *malloc();
        Fmatrix x;
        int mn, namlen;

        if (fread((char *)&x, sizeof(Fmatrix), 1, fp) != 1) {
             return(1);
        }
        *type = x.type;
        *mrows = x.mrows;
        *ncols = x.ncols;
        *imagf = x.imagf;
        namlen = x.namlen;
        mn = x.mrows * x.ncols;

        if (fread(pname, sizeof(char), namlen, fp) != namlen)  return(1);
        if (!(*preal = (double *)malloc(mn*sizeof(double)))) {
             printf("\nError: Variable too big to load\n");
             return(1);
        }
        if (fread(*preal, sizeof(double), mn, fp) != mn) {
             free(*preal);
             return(1);
        }

        if (x.imagf) {
             if (!(*pimag = (double *)malloc(mn*sizeof(double)))) {
                  printf("\nError: Variable too big to load\n");
                  free(*preal);
```

```
                        return(1);
                }
                if (fread(*pimag, sizeof(double), mn, fp) != mn) {
                        free(*pimag);
                        free(*preal);
                        return(1);
                }
        }
        return(0);
}
```

```
/*      file:           f18l_modelN.h
 *      date:           8 Feb 1992
 *      revised:        28 March 1992
 *      programmer:     Shahar Dror
 *
 *      This file contains the definitions and the model of the f18 fighter
 *      aircraft. Also it defines initial conditions.
 *
 *      The names of the functions appearing in this file must remain unchanged!
 *
 *********************************************************************************/

/* All definitions in this box MUST appear!! If don't care set to default=1(one)
unless otherwise specified by: "!def=..." */
/*==========\/==========\/==========\/==========\/==========\/==========\/====*/

#define plant_input_order  6        /* order of plant inputs vector */
#define plant_output_order 3        /* order of plant outputs */
#define in_history         3        /* # of delayed input parameters */
#define out_history        3        /* # of delayed output parameters */
#define Nmodels            2        /* # of models to train on(>=1). !def=1
/*==========/\==========/\==========/\==========/\==========/\==========/\====*/

/*@ The following definitions are in effect ONLY in this header file. The     @
  @ names can be changed, after checking that the desired name does not exist@
  @ in other files that are in use together with this header-file !!!        @*/

#define      plant_order  25      /* order of plant model      */
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/

 /* Declarations */
/* ============== */
/* The values under declarations MUST appear!! If "don't care" set to default,
which is 0 (zero) in the correct type, unless otherwise specified by "def=..."*/
/*==========\/==========\/==========\/==========\/==========\/==========\/====*/

double ts=0.02;             /* Sampling interval for discretization. def=1.0 */
double tau=17.74;   /* Characteristic(longest) time constant [sec](Phugoid)*/
/* Frequency band of interest [Hz] in the current model */
double Fmax=5.0;    /* Upper frequency (used for swept wave test) */
double Fmin=0.005;  /* Lowest frequency (used for swept wave test) */
double excite[plant_input_order]={0.0};/* Plant input commands = NN outputs */
double bias_I[Nmodels][plant_input_order]={0.0}; /*Plant input bias matrix*/
/*==========/\==========/\==========/\==========/\==========/\==========/\====*/

/*@ The following declarations are in effect ONLY in this header file. The    @
  @ names can be changed, after checking that the desired name does not exist@
  @ in other files that are in use together with this header-file !!!        @*/

double excite_amplitude[plant_input_order] =  /* Scaling the input commands */
           {1.0,1.0,1.0,1.0,1.0,1.0};
/* Initial conditions. If !=0 put them here! */
double plant_state_ic[plant_order]={0.0};/* Plant initial conditions vector */

double plant_A[Nmodels][plant_order*plant_order];/* Plant dynamic matrix */
double plant_B[Nmodels][plant_order*plant_input_order];/* Plant control matrix*/
double plant_C[Nmodels][plant_output_order*plant_order];/*Plant measure matrix*/
double plant_D[Nmodels][plant_output_order*plant_input_order];/*Plant ff matrx*/
double bias_O[Nmodels][plant_output_order]={0.0};/*Plant output bias matrix*/

/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/

load_model(mnr)     /* Loading matrices of current model to work with */
int mnr;      /* Model number, 1,2,...,Nmodels */
{
double *xr,*xi;     /* Real and imaginary parts of the loaded matrix */
int type, mrows, ncols, imagf;
int ret_val=UIO_OK;
int i,j;       /* Indexes */
char name[20];      /* Name of loaded matrix. Not used. */
char intstr[10];    /* String to hold integer to string conversion */
char matfile[25];   /* string to .mat file name for model matrices */
FILE *ck_open_file(),*fp;
```

```
sprintf(intstr,"%i",mnr);
strcat(intstr,".mat");     /* Create extension to model matrices as XXX#.mat */

strcpy(matfile,"zFld");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL) return(UIO_ERROR);
loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);  fclose(fp);
          if((mrows != plant_order) || (ncols != plant_order)){
          PutFmt("\n!!Plant mismatch. plant_A[model %i]is[%ix%i], plant_order=%i",
          mnr,mrows,ncols,plant_order);
          PutStr("\nError in load_model(). Press RETURN to stop and exit");
          GetStr();     return(UIO_ERROR);
          }
          for(i=0;i<mrows*ncols;plant_A[mnr-1][i]=xr[i],i++);

strcpy(matfile,"zGld");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL) return(UIO_ERROR);
loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);  fclose(fp);
          if((mrows != plant_order) || (ncols != plant_input_order)){
          PutFmt("\n!Plant mismatch. plant_B[model%i]is[%ix%i],but set to [%ix%i]"
          ,mnr,mrows,ncols,plant_order,plant_input_order);
          PutStr("\nError in load_model(). Press RETURN to stop and exit");
          GetStr();     return(UIO_ERROR);
          }
          for(i=0;i<mrows*ncols;plant_B[mnr-1][i]=xr[i],i++);

strcpy(matfile,"zHld");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL) return(UIO_ERROR);
loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);  fclose(fp);
          if((mrows != plant_output_order) || (ncols != plant_order)){
          PutFmt("\n!Plant mismatch.plant_C[model %i]is[%ix%i],but set to [%ix%i]"
          ,mnr,mrows,ncols,plant_output_order,plant_order);
          PutStr("\nError in load_model(). Press RETURN to stop and exit");
          GetStr();     return(UIO_ERROR);
          }
          for(i=0;i<mrows*ncols;plant_C[mnr-1][i]=xr[i],i++);

strcpy(matfile,"zDld");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL) return(UIO_ERROR);
loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);  fclose(fp);
          if((mrows != plant_output_order) || (ncols != plant_input_order)){
          PutFmt("\n!Plant mismatch.plant_D[model %i]is[%ix%i],but set to [%ix%i]"
          ,mnr,mrows,ncols,plant_output_order,plant_input_order);
          PutStr("\nError in load_model(). Press RETURN to stop and exit");
          GetStr();     return(UIO_ERROR);
          }
          for(i=0;i<mrows*ncols;plant_D[mnr-1][i]=xr[i],i++);

strcpy(matfile,"zBil");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL){
          PutStr("\nAssuming no input-biasing^G");
          printf("\nAssuming no input-biasing^G");
          }
          else{
          loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);    fclose(fp);
                if(!(((mrows==plant_input_order)||(ncols == plant_input_order))
                      && ((mrows == 1)||(ncols == 1)))){
            PutFmt("\n!Bias mismatch.bias_I[model %i]is[%ix%i],but set to [%ix%i]"
                ,mnr,mrows,ncols,1,plant_input_order);
                PutStr("\nError in load_model(). Press RETURN to stop & exit");
                GetStr();     return(UIO_ERROR);
                }/*end if*/
                for(i=0;i<mrows*ncols;bias_I[mnr-1][i]=xr[i],i++);
          }/*end else*/

strcpy(matfile,"zBol");strcat(matfile,intstr);
          if((fp=ck_open_file(matfile,"r"))==NULL){
          PutStr("\nAssuming no output-biasing^G");
          printf("\nAssuming no output-biasing^G");
          }
          else{
          loadmat(fp,&type,name,&mrows,&ncols,&imagf,&xr,&xi);    fclose(fp);
                if(!(((mrows==plant_output_order)||(ncols ==plant_output_order))
                      && ((mrows == 1)||(ncols == 1)))){
            PutFmt("\n!Bias mismatch.bias_O[model %i]is[%ix%i],but set to [%ix%i]"
          ,mnr,mrows,ncols,1,plant_output_order);
          PutStr("\nError in load_model(). Press RETURN to stop & exit");
          GetStr();     return(UIO_ERROR);
          }/*end if*/
          for(i=0;i<mrows*ncols;bias_O[mnr-1][i]=xr[i],i++);
     }/*end else*/

return(ret_val);
}

PLANT(ic,mnr,Plant_input,Plant_out)/*Mathematical description of plant output */
int ic;       /* (Initial) conditions type indicator. IClearn,ICtest, TEST or LEARN */
int mnr;      /* Model # in effect for current desired output calculation */
              /* 1 - non damaged, original. >=2 - impaired, other */
double Plant_input[plant_input_order];         /* Control inputs vector */
double Plant_out[plant_output_order];  /*f18 measurements+model coding */
{
/* This function is written in details in order to be able to put in any non-linearities at any
point in the calculation*/
static int mnr_last=1;     /* previos model number. Initially the first. */
static double plant_state[Nmodels][plant_order];       /* f18 state-vectors */
double plant_msr_state[plant_output_order];            /* f18 msr vector */
double wv[plant_order];    /*working vector*/
int i,j;
/* First X(k+1) is calculated using X(k), U(k), then - Y(k+1) using X(k+1). */
/* For initial conditions don't update the states */
     if(ic==IClearn || ic==ICtest){
          for(i=0;i<plant_order;i++)
          plant_state[mnr-1][i]=( (ic==ICtest) ? 0.0 : plant_state_ic[i]);
     goto ICjump;
     }
/* For continuous RBS input (=1) start new model with last state value of
   previous model. */
     if(ic==TEST1 && mnr!=mnr_last){
          for(i=0;i<plant_order;i++)
          plant_state[mnr-1][i]=plant_state[mnr_last-1][i];
     mnr_last=mnr;
     }
/* Plant state update*/
     for(i=0;i<plant_order;i++){
     wv[i]=0.0;
          for(j=0;j<plant_order;j++)                         /* Ax */
          wv[i]+=(plant_A[mnr-1][i+plant_order*j]*plant_state[mnr-1][j]);
          for(j=0;j<plant_input_order;j++)                 /*+B(u+bi) */
          wv[i]+=(plant_B[mnr-1][i+plant_order*j]
                *(Plant_input[j]+bias_I[mnr-1][j])*excite_amplitude[j]);
     }
     for(i=0;i<plant_order;plant_state[mnr-1][i]=wv[i],i++);         /* X(k+1) */

ICjump:

/* Plant measurement update*/
     for(i=0;i<plant_output_order;i++){                     /* Cx+Du+bo */
     Plant_out[i]=0.0;
          for(j=0;j<plant_order;j++)
          Plant_out[i]+=
          (plant_C[mnr-1][i+plant_output_order*j]*plant_state[mnr-1][j]);
/* in my case = 0           for(j=0;j<plant_input_order;j++)
          Plant_out[i]+=(plant_D[mnr-1][i+plant_output_order*j]
          *(Plant_input[j]+bias_I[mnr-1][j])*excite_amplitude[j]);*/
     Plant_out[i]+=bias_O[mnr-1][i];
     }
}
```

```
/***************************************************************************/
/*     File:  net_paramNtp.h                                               */
/*     Author:        Shahar Dror                                          */
/*     Date:  9 Mar 92                                                     */
/*     Revised:17 Jun 92                                                   */
/*     Subject:       This header file contains declarations and           */
/*            definitions of parameters that are used in the USRIO.        */
/*            It comes before net_util.h and after ref_modelN.h,*/
/*             and uses parameters/values declared there.                  */
/*                        This header file also contains functions*/
/*            used by the USRIO program for its execution. Parameters      */
/*            declared here are global, and accessible to every            */
/*            function in the USRIO, and also to every function            */
/*            hereafter. Declarations and definitions here should not */
/*            be deleted.                                                  */
/*            This file creates the base parameters that describe the      */
/*            problem being worked on. Their VALUES may be altered         */
/*            (but not their names). Default values (i.e if "don't-        */
/*            care") are zero (0), unless otherwise specified. 3/6/92      */
/*                                                                         */
/***************************************************************************/

/* Essential definitions. Provide work base for the current problem. !Default=0*
 * The variables are usually self-explanatory by their names.              */
/*=========================================================================*/
#if Task==classification          /* Set up classification mode */
#define forced_excitation_length              plant_input_order
#define unforced_excitation_length            plant_output_order
#define type_length                           0
#define      excitation_length  (forced_excitation_length+unforced_excitation_length)
#define      regressed_input_length           excitation_length
#define      input_lay_length                 regressed_input_length
#define      regressed_input_history_length   regressed_input_length*in_history
#define      dynamic_output_length            0
#define      regressed_output_length          0
#define      static_output_length             Nmodels
#define      output_lay_length                Nmodels
#define      regressed_output_history_length  regressed_output_length*out_history
#define effector_lay_length                   0
#endif

#if Task==emulation        /* Set up emulation mode */
#define forced_excitation_length              plant_input_order
#define unforced_excitation_length            0
#define type_length                           Nmodels
#define excitation_length                     forced_excitation_length
#define regressed_input_length                forced_excitation_length
#define input_lay_length                      regressed_input_length+type_length
#define regressed_input_history_length        regressed_input_length*in_history
#define dynamic_output_length                 plant_output_order
#define static_output_length                  0
#define regressed_output_length               plant_output_order
#define output_lay_length                     plant_output_order
#define regressed_output_history_length       regressed_output_length*out_history
#define effector_lay_length                   0
#endif

#if Task==combined_cl_em   /* Set up classification and emulation mode */
#define forced_excitation_length              plant_input_order
#define unforced_excitation_length            0
#define type_length                           Nmodels
#define excitation_length                     forced_excitation_length
#define regressed_input_length                forced_excitation_length
#define input_lay_length                      regressed_input_length+type_length
#define regressed_input_history_length        regressed_input_length*in_history
#define dynamic_output_length                 plant_output_order
#define static_output_length                  Nmodels
#define regressed_output_length               plant_output_order
#define output_lay_length                     plant_output_order+type_length
#define regressed_output_history_length       regressed_output_length*out_history
#define effector_lay_length                   0
#endif
```

```
#if Task==controlling           /* Set up controlling mode */
#define model_input_length                  6
#define model_output_length                 plant_output_order
#define forced_excitation_length            plant_input_order
#define unforced_excitation_length          0
#define type_length        (model_input_length+model_output_length+Nmodels)
#define excitation_length                   forced_excitation_length
#define regressed_input_length              excitation_length
#define input_lay_length                    regressed_input_length+type_length
#define regressed_input_history_length      regressed_input_length*in_history
#define dynamic_output_length               plant_output_order
#define static_output_length                0
#define regressed_output_length             plant_output_order
#define output_lay_length                   plant_output_order
#define regressed_output_history_length     regressed_output_length*out_history
#define effector_lay_length                 model_input_length
#endif

#if Task==cntrlThruPlant        /* Set up controlling mode */
#define ref_in_length                       ref_input_order
#define ref_out_length/*eq plant_out*/      ref_output_order
#define plant_in_length/*  ||    */         plant_input_order
#define plant_out_length/*eq ref_out*/      plant_output_order
#define forced_excitation_length            ref_input_order
#define unforced_excitation_length          ref_output_order
#define type_length                         (plant_input_order+plant_output_order)
#define excitation_length  (forced_excitation_length+unforced_excitation_length)
#define regressed_input_length              (excitation_length+type_length)
#define input_lay_length                    (excitation_length+type_length)
#define regressed_input_history_length      regressed_input_length*in_history
#define dynamic_output_length               plant_input_order
#define static_output_length                0
#define regressed_output_length             0
#define output_lay_length                   dynamic_output_length
#define regressed_output_history_length     regressed_output_length*out_history
#define effector_lay_length                 output_lay_length
#endif


/*    If the regressed values are not consequent and as described here,*
 *  indicate the sorting in the functions regress_input_sort(), and    *
 *  regress_output_sort(). These appear down in this header file.      *
 *     The user must take care and connect the relevant input nodes to *
 *  the command regression layer in the network (while building it)!!  */
/*=====================================================================*/

#define RbsUd       4           /* RBS update divisor */
#define NErrConv    20          /* # of converged RMS error for best network check */
#define NetSaveMode 0           /* 0=binary, 1=ascii, 2=annotated, 3=screen dump */
#define nInstElmnt  8190        /* max # of elements sent by an instrument */
#define maxInstName 25          /* Max length of instrument name */
#define Ninput_types        6   /* # of input types (see *input_name -1) */
#define maxNsweep_change   700  /* Max # of sign switching for H->L swept wave*/
double dpl[]={0.5,1.5,2.5,5.0}; /* dipole switching times for 0,+1,-1,0,end. */
double convergence=2e-3;        /* Total error RMS convergence criterion */
double h=0.01;                  /* Spatial step size for Jacobian calculation */
float min_lim=-2.0,max_lim=2.0; /* Limit values for network outputs */
char NNrslt[20]="NNout.dat";    /* file name of NN results in TEST mode*/
char DesO[20]="DesOut.dat";     /* file name of desired outputs in TEST mode*/
char Inp[20]="Input.dat";       /* file name of inputs in TEST mode*/
char *input_name[]={            /* !!!check (#define Ninput_types) up here!!! */
                    "Illegal Input",                /* input=0 */
                    "Random Binary",                /* input=1 */
                    "Sequential Random Binary",     /* input=2 */
                    "Swept H->L Square Wave",       /* input=3 */
                    "Swept L->H Square Wave",       /* input=4 */
                    "Swept L->H Sine Wave",         /* input=5 */
                    "1 sec dipole"};                /* input=6 */
int input=1;                /* Input type indicator, initially set to RBS (=1) */
int rmRC=1000;              /* Divisor of RBS random TEST cycles model switching */
int maxSW=2,minSW=1;        /* Divisor,bias of RBS random cycles model switching */
int nRTC,ResetTC=10000;     /* Reset-training-cycles mode divisor */
```

```
/* Place here global parameters that are needed for the specific implementation.
      Check that they don't exist in the program, including all include files.
      These parameters are to be used within functions present in this
      header file (net_paramXXX.h).          */
/*-----------V-----------V-----------V-----------V-----------V----------*/
#if Task==cntrlThruPlant
double refout[Nref_models][ref_out_length];
double plantout[Nmodels][plant_out_length];
double outerr[plant_out_length];
double W[plant_out_length]={1.0,1.0,1.0};     /* Output weighting vector */
#endif
/*-----------^-----------^-----------^-----------^-----------^----------*/

regress_input_sort(RegInSort)     /* Input sorting function */
int RegInSort[regressed_input_length];
{
int i;        /* Index counter */
/* Default sorting is in sequential order */
      for(i=0;i<regressed_input_length;RegInSort[i]=i,i++);
}

regress_output_sort(RegOutSort)    /* Output sorting function */
int RegOutSort[regressed_output_length+1];
{
int i;        /* Index counter */
/* Default sorting is in sequential order */
      for(i=0;i<regressed_output_length;RegOutSort[i]=i,i++);
}

int Limits(mnr,out_val)    /* Returns 1 if inside limits. Else returns 0 */
/* User can write any limit function, per each output, or common to all outputs,
with min/max or abs values, etc. */
int mnr;      /* Model # in effect for current Jacobian calculation
                     1 - non damaged, original. >=2 - other */
float out_val[output_lay_length];
{
int i;                 /* counter */
int lim;      /* if=1, within limits, if=0, out of limits */
      for(i=0;(i<output_lay_length && lim!=0);i++)
      lim=((out_val[i]>=min_lim && out_val[i]<=max_lim)?1:0);
return(lim);
}

jacobian(mnr,plant_input,plant_output,jacb)      /* Jacobian calculation */
double jacb[plant_output_order*plant_input_order];     /* Jacobian matrix */
double plant_input[plant_input_order];                 /* Control inputs vector */
double plant_output[plant_output_order];   /* Current plant outputs vector */
int mnr;      /* Model # in effect for current Jacobian calculation
                     1 - non damaged, original. >=2 - other */
{
static double old_plant_input[Nmodels][plant_in_length]={0.0};/* Last inputs */
double dh;    /* Spatial increment including sign */
double plant_out[plant_output_order];         /* plant output @ perturbated inputs */
int i,j;      /* indexes */

      for(i=0;i<plant_input_order;i++){
      /* Choose the sign of increment to ensure staying in the domain */
      dh=copysign(h,(old_plant_input[mnr-1][i]-plant_input[i]));
      plant_input[i]+=dh;
      PLANT(JACOB,mnr,plant_input,plant_out);
             for(j=0;j<plant_output_order;j++){
             jacb[i*plant_output_order+j]=(plant_out[j]-plant_output[j])/dh;
             }
      old_plant_input[mnr-1][i]=(plant_input[i]-=dh);
      }
}

desired_output(ic,mnr,Plant_input,dso)        /* Get desired output */
int ic;       /* (Initial) conditions type indicator. IClearn, ICtest, TEST, LEARN,
              TEST1 or JACOB to indicate Jacobian calculation. */
int mnr;      /* Model # in effect for current desired output calculation */
              /* 1 - non damaged, original. >=2 - impaired, other */
double Plant_input[forced_excitation_length];         /* Control inputs vector */
double dso[output_lay_length];
```

```
{
extern double Effector[Nmodels][effector_lay_length+1];
double jac[plant_output_order*plant_input_order];     /* Jacobian matrix */
extern int error_flag;
int i,j;      /* Index counter */
      if(Task==emulation)          PLANT(ic,mnr,Plant_input,dso);
      else if(Task==combined_cl_em){
                                   PLANT(ic,mnr,Plant_input,dso);
            for(i=0;i<type_length;i++)
                  dso[dynamic_output_length+i]=(i==(mnr-1)?1.0:-1.0);
                                   BCD(mnr,&dso[dynamic_output_length]);
      }
      else if(Task==controlling){
      REF(ic,mnr,Plant_input,dso);
      PLANT(ic,mnr,&Effector[mnr-1][0],&plantout[mnr-1][0]);
      }
      else if(Task==cntrlThruPlant){
      /* In this implementation, for every plant there is a corresponding
      reference. If the situation is not so, there should be a matching
      between the plant # (mnr) and the value sent to REF().  */
      REF(ic,mnr,Plant_input,&refout[mnr-1][0]);
      PLANT(ic,mnr,&Effector[mnr-1][0],&plantout[mnr-1][0]);
            for(i=0;i<plant_out_length;i++)
            outerr[i]=refout[mnr-1][i]-plantout[mnr-1][i];

      jacobian(mnr,&Effector[mnr-1][0],&plantout[mnr-1][0],jac);
            for(i=0;i<plant_in_length;i++){
            dso[i]=0.0;
                  for(j=0;j<plant_out_length;j++)
                  dso[i]+=(W[j]*outerr[j]*jac[i*plant_out_length+j]);
            dso[i]+=Effector[mnr-1][i];
            }
      }

/* User put before this line more tasks if desired. The "else" must be last! */
      else{
      PutFmt("\n!!No desired_output() for specified Task (= %i)",Task);
      printf("\n!!No desired_output() for specified Task (= %i)",Task);
      error_flag=SET;      BEEP;
      }
}

unforced_excitation(ic,mnr,Plant_input,ue)     /* Get unforced excitation */
int ic;        /* (Initial) conditions type indicator. ICLearn, TEST or LEARN */
int mnr;       /* Model # in effect for current desired output calculation */
               /* 1 - non damaged, original. >=2 - impaired, other */
double Plant_input[forced_excitation_length];        /* Control inputs vector */
double ue[unforced_excitation_length+1];
{
static double ueOld[Nmodels][unforced_excitation_length+1];
extern double desoutOld[Nmodels][output_lay_length];
extern int error_flag;
int i;
      if(Task==classification){
       for(i=0;i<unforced_excitation_length;ue[i]=ueOld[mnr-1][i],i++);
      PLANT(ic,mnr,Plant_input,&ueOld[mnr-1][0]);
      }
      else if(Task==controlling){/* The prior value of the reference model */
            /*for(i=0;i<unforced_excitation_length;i++)
            ue[i]=desoutOld[mnr-1][i];*/
      }
      else if(Task==emulation);  /* No unforced_excitation in emulation*/
      else if(Task==cntrlThruPlant){

            for(i=0;i<unforced_excitation_length;i++){
            /*    if(ic==ICLearn || ic==LEARN)*/
                  ue[i]=refout[mnr-1][i];   /* it is the old value */
            /*    if(ic==ICtest || ic==TEST)
                  ue[i]=plantout[mnr-1][i];/* it is the old value */
            }

      }
/* User put before this line more tasks if desired. The "else" must be last! */
         else{
         PutFmt("\n!!No unforced_excitation() for specified Task (= %i)",Task);
         printf("\n!!No unforced_excitation() for specified Task (= %i)",Task);
         error_flag=SET;     BEEP;
         }
}


Type(ic,mnr,nnout,effect_out,mtype)                    /* Model type identification */
int ic;       /* (Initial) conditions type indicator. IClearn, TEST or LEARN */
int mnr;                  /* The model # */
double nnout[output_lay_length];  /* Network output. Not always needed. */
double effect_out[effector_lay_length+1];/* Effector value. Not always needed.*/
double mtype[type_length+1];       /* Its representation */
{
extern int error_flag;
int i;        /* Index counter */
         if(Task==emulation)
                  for(i=0;i<type_length;mtype[i]=(i==(mnr-1)?1.0:-1.0),i++);
         else if(Task==controlling){
                  for(i=0;i<effector_lay_length;mtype[i]=effect_out[i],i++);
                  for(i=0;i<output_lay_length;i++)
                                   mtype[i+effector_lay_length]=nnout[i];
                  for(i=0;i<Nmodels;i++)
                  mtype[i+type_length-Nmodels]=(i==(mnr-1)?1.0:-1.0);
         }
         else if(Task==cntrlThruPlant){

                  for(i=0;i<effector_lay_length;i++){
                  mtype[i]=effect_out[i];
                  }
                  for(i=0;i<plant_out_length;i++) /* it is the old value */
                  mtype[i+effector_lay_length]=plantout[mnr-1][i];

         }

/* User put before this line more tasks if desired. The "else" must be last! */
         else{
         PutFmt("\n!!No Type() for specified Task (= %i)",Task);
         printf("\n!!No Type() for specified Task (= %i)",Task);
         error_flag=SET;     BEEP;
         }
}
```

```
/***********************************************************************/
/*      File: net_util.h                                               */
/*      Author:        Shahar Dror                                     */
/*      Date: 9 Mar 92                                                 */
/*      Revised:                                                       */
/*      Subject:       This header file contains declarations and      */
/*              definitions of parameters that are used in the USRIO.  */
/*              It comes after net_param.h, and uses parameters/values */
/*              declared there. This header file also contains functions*/
/*              used by the USRIO program for its execution. Parameters */
/*              declared here are global, and accessible to every      */
/*              function in the USRIO, and also to every function      */
/*              hereafter. Declarations and definitions here should not */
/*              be deleted.                                            */
/*                                                                     */
/***********************************************************************/


#define RBS   (2.0*(random()&01)-1.0)                 /* Random Binary Sequence */
#define DIPOLE(T) (T>=dpl[0]&&T<dpl[1])?1.0:((T>=dpl[1]&&T<dpl[2])?-1.0:0.0)
#define Large_number        (int)(exp2(31.0)-1)       /* Large integer # */
#define PI 3.14159265358979          /* The constant pi */
double desout[Nmodels][output_lay_length];    /* Desired output */
double desoutOld[Nmodels][output_lay_length];         /* Last desired output */
double unfexc[Nmodels][unforced_excitation_length+1]={0.0};/*Unforced excita'n*/
double ComReg[Nmodels][regressed_input_history_length];/*Command regres output*/
double OutReg[Nmodels][regressed_output_history_length+1];/*Out-regress output*/
double Effector[Nmodels][effector_lay_length+1]={0.0};/*Effector/Controler out*/
double input_data[Nmodels][input_lay_length]={0.0};   /* Input-layer data */
double err_RMS,max_RMS=0.0,min_RMS=1.0;        /* Relative mean square output error */
double dmin,df,f,T; /* log10(Fmin), deltaF,F and Time for swept wave  */
double dflog,dflin,fflog,flog,flin;            /* Frequences  for swept wave  */
double one=1.0;             /* unit value for H->L swept square wave */
double min_sum,sum[NErrConv+1]={0.0};          /* RMS error convergence parameters */
char *BestNetName;  /* file name of the best network results */
int effector_lay=7; /* Controller output layer # (initial guess) */
int rand_inputs[forced_excitation_length];/*Array of random unrepeated inputs */
int rand_models[Nmodels+1];         /* Array of randomly unrepeated  model #s. */
int RIsort[regressed_input_length];             /*Ordering  input units for regression*/
int ROsort[regressed_output_length+1];          /*Ordering output units for regression*/
int Tau;                /* the value "tau" measured in ts units ==tau/ts*/
int nlayp,ninp,noutp;       /* Pointers to: # of layers,# of input/output nodes */
int history;            /* # of cycles w/o learning to fill regression layers */
int inTest;             /* # of input under test, or training input in LEARN  */
int model_nbr;              /* # of model under test, or training */
int mRC;                /* minimum Recall/test/learn-switching Cycles */
int MinErTC,MaxErTC;        /* min/max error training/testing cycles indicator */
int conv_test[2]={0,1};     /* Convergence test counters */
int Nsweep_change,sweep_count;     /* High->Low swept square wave counters */
int sweep_change_times[maxNsweep_change];     /* H-L switching times vector */
int netSave_flag=RESET;     /* Flag when (RESET)/SET (don't)/save best network */
int TestDataSave_flag=RESET;       /* Flag (RESET)/SET (don't)/save TEST results */
int error_flag=RESET;       /* Flag when (RESET)/SET an error (didn't) occurred */
int intrrpt_flag=SET;       /* Flag when (RESET)/SET run was (not) interrupted */
int A1model_flag=SET;       /* Flag when (RESET)/SET (more than) one model active */
int Rstart_flag=RESET;      /* Flag when (RESET)/SET (not) passed UIO_Recall_Start*/
int ResetTrain_flag=SET;    /* Flag when (RESET)/SET (dont) use Reset-trainig mode*/
int ResetLimit_flag=RESET; /*(RESET)/SET (dont) Reset when output exceed limits*/
int netReset_flag=RESET;    /*(RESET)/SET (dont) Reset network */
int efLay_flag=RESET;       /* Flag when (RESET)/SET (dont) recheck effector_lay #*/
FILE *NNo;    /* NN output file in TEST mode */
FILE *DESo;  /* Desired output file in TEST mode */
FILE *InPut; /* Input data file in TEST mode */


/* In order to prevent compiling errors */
#ifndef Nref_models
#define Nref_models 0
load_ref_model(){}
#endif
```

```
int check_net()
/* This routine checks that the # of input/ouput PEs match the desired in the USRIO program. If
there is a match the routine returns UIO_OK, on failure it returns UIO_ERROR */
{
/* Network parameters */
int   ltype;          /* Network parameters */
static int nlays=0; /* # of layers reminder */
char  *csp,*netnp;  /* Network pointers*/
char *sp;           /* pointer to a string */
long  ctrp[8];      /* An array of 8 to interact with control-strategy*/

      /* Read network parameters */
      ug_rdnetinf( &nlayp, &ninp, &noutp, &ltype, &csp, &netnp );
/* Check if the built network fits the current plant, control and history */
       if(ninp != input_lay_length){
       PutFmt("\n!! The network doesn't match. nPEs_in=%i, but has to be :%i",
       ninp,input_lay_length);
       PutStr("\nPress RETURN to stop and exit"); GetStr();
       return(UIO_ERROR);
       }
       if(noutp != output_lay_length){
       PutFmt("\n!! The network doesn't match. nPEs_out=%i, but has to be :%i",
       noutp,output_lay_length);
       PutStr("\nPress RETURN to stop and exit"); GetStr();
       return(UIO_ERROR);
       }
       /* Check consistency between parameters and network structure */
       if(nlayp<5 && regressed_output_length>0){
       PutFmt("\n!! OutReg layer requested. At least 5 layers needed. nLays=%i"
       ,nlayp);     BEEP;
       PutStr("\nPress RETURN to stop and exit"); GetStr();
       return(UIO_ERROR);
       }
/* Set Effector output layer #, and specify it to the control strategy via AUX2
 counter */
       if(effector_lay_length>0 && (nlays!=nlayp || efLay_flag==SET)){
       PutFmt("\nEnter effector-layer # (input_lay=0) [%i] ",effector_lay);
       sp=GetStr();
              if(*sp) sscanf(sp,"%i",&effector_lay);
       ug_ctrio(ctrp,0);   /* read counters content */
              if(effector_lay==(nlayp-1))      ctrp[4]=0;
              else ctrp[4]=(long)effector_lay;/* Set Aux2= effector-layer # */
       ug_ctrio(ctrp,1);   /* write counters content */
       nlays=nlayp;        /* Set # of layers reminder to # of layers */
       efLay_flag==RESET;  /* Reset effector_lay # checking */
       }

/* save network name for storing best network results */
BestNetName=strcat(netnp,"Bst");

return(UIO_OK);
}

void set_net()
{
/* Setting and initializing different network and other related parameters */
/* THIS FUNCTION MAY BE EDITED (BY ADDING!!) BY USER */

Tau=(int)(ceil(tau/ts));  /* Setting time constant-tau in units of ts */
       /* Sort the # if the output units to be regressed if not consecutive */
regress_output_sort(ROsort);
       /* Sort the # if the input units to be regressed if not consecutive */
regress_input_sort(RIsort);
}


FILE *ck_open_file(fpn,type)
char *fpn, *type;
/*     Opens a file for r/w .
       fpn - is the name of the file to be open.
       type - is "r" for read, "w" for write.
       On success, it return the file pointer. If the file can not be open it
       gives a statement and returns NULL.    */
{
```

```
FILE *fp;
        if((fp=fopen(fpn,type))==NULL){
        printf(" Can't open (for %s) file: %s", type,fpn);
        PutFmt(" Can't open (for %s) file: %s", type,fpn);
        }
return(fp);
}

int open_TEST_files()      /* Open files for saving TEST results */
{
char *sp;

PutFmt("\nEnter file name to save Inputs [%s] ",Inp);
sp=GetStr();
        if(*sp) sscanf(sp,"%s",Inp);
        if((InPut=ck_open_file(Inp,"w"))==NULL) return(UIO_ERROR);
PutFmt("\nEnter file name to save NN output [%s] ",NNrslt);
sp=GetStr();
        if(*sp) sscanf(sp,"%s",NNrslt);
        if((NNo=ck_open_file(NNrslt,"w"))==NULL) return(UIO_ERROR);
PutFmt("\nEnter file name to save desired output [%s] ",DesO);
sp=GetStr();
        if(*sp) sscanf(sp,"%s",DesO);
        if((DESo=ck_open_file(DesO,"w"))==NULL) return(UIO_ERROR);
return(UIO_OK);
}


void close_files()
/*      Closes all opened files     */
{
fclose(InPut);       /* Closing input file (if exists) */
fclose(NNo); /* Closing NN output file (if exists) */
fclose(DESo);        /* Closing desired output file (if exists) */
return;
}

void rand_array(N,rnd_vec)
/* This function generates unrepeated (N) integer random #s valued 1 to N. */
int N;         /* # of elements in the random vector.(==forced_excitation_length)*/
int *rnd_vec;        /* Array containing unrepeated random #s */
{
int i,j;
rnd_vec[0]=(random()%N)+1;
        for(j=1;j<N;j++){
                do(    /* generate new non repeatetive random input */
                rnd_vec[j]=(random()%N)+1;
                        for(i=0;(i<j && rnd_vec[j] != rnd_vec[i]);i++);
                } while(i != j);
        }
}

BCD(N,Nbcd)  /* Convert N to its Binary Coded Decimal representation */
int N;                     /* The # to be converted */
double Nbcd[type_length+1];       /* Its BCD representation */
{
int i;         /* index */
int nc;        /* copy of N */
nc=2*(N-1);
        for(i=0;i<type_length;Nbcd[i]=(double)((nc=nc/2)%2)*2-1,i++);
}


/* From here on are function of different input-commands for testing.  */
/*____________________________________________________________________*/

int sweep_HL(tmin,tmax,s,ltn)
int tmin,tmax,s;
int *ltn;
{
char *sp;
int i,lsum,sum,n=2;
double dt,tn[maxNsweep_change];
tn[0]=ltn[1]=tmin;
```

```
        do{
                if((++n)>=maxNsweep_change) {
                PutFmt(
        "\nToo few H->L sweep changing points. Increase maxNsweep_change=%i",
                maxNsweep_change);
                PutStr("\nPress RETURN to stop and exit"); sp=GetStr();
                error_flag=SET;
                return(UIO_ERROR);
                }
        lsum=sum;
        sum=ltn[1];
        dt=((double)(1.0/tmax)-(double)(1.0/tmin))/(n-1);
                for(i=1;i<n;i++){
                ltn[i+1]=nint(tn[i]);
                tn[i]=1.0/(1.0/tn[i-1]+dt);
                sum+=nint(tn[i]);
                }
        } while(sum<s);
ltn[--n]+=(s-lsum);
ltn[0]=n;
return(UIO_OK);
}

double SweptSquareWaveH2L()
/* Generating alternating swept square wave starting with high frequency
down to the low frequency*/
{
        if((sweep_count++)==sweep_change_times[Nsweep_change+1]){
        Nsweep_change=(++Nsweep_change)%sweep_change_times[0];
        sweep_count=1;      /* renew counting for the next step */
        one=-one;     /* switching state +/- 1 */
        }
return(one);
}

double SweptSquareWaveL2H(cycle)
int cycle;
/* Generating alternating swept square wave starting with low frequency
up to the high frequency */
{
/* Shaping the spectrum of the swept wave */
fflog+=dflog;                 /* Incrementing logarithmic frequency */
flog=exp10(fflog);  /* Logarithmic frequency */
flin+=dflin;         /* Incrementing linear frequency */
df=pow((double)(mRC-1-cycle%mRC)/(double)(mRC-1),3.0)*60.0;
f=(df*flog+flin)/(df+1.0);
T+=ts;                        /* Incrementing Time */
return(copysign(1.0,sin(2*PI*f*T)));
}

double SweptSineWaveL2H(cycle)
int cycle;
/* Generating alternating swept sine wave starting with low frequency
up to the high frequency */
{
/* Shaping the spectrum of the swept wave */
fflog+=dflog;                 /* Incrementing logarithmic frequency */
flog=exp10(fflog);  /* Logarithmic frequency */
flin+=dflin;         /* Incrementing linear frequency */
df=pow((double)(mRC-1-cycle%mRC)/(double)(mRC-1),3.0)*60.0;
f=(df*flog+flin)/(df+1.0);
T+=ts;                        /* Incrementing Time */
return(sin(2*PI*f*T));
}
```

```
/*     file:         window_dat.h
 *     date:         14 Oct 1991
 *     revised:      14 Oct 1991
 *     programmer:   Shahar Dror
 *
 *     This file contains the definitions for windowing and plotting relative
 *      errors of the output layer.
 *
 ******************************************************************************/
#define Ytop 800           /* Topmost window place */
#define Ybot 450           /* Bottommost window place */
#define Yspace 10          /* Horizontal window spacing */
#define Xstart 10          /* Left-most window place */
#define Xmsg Xstart+100    /* Left-most message window place */
#define Ymsg Ytop          /* Bottommost message window place */
#define Hmsg 20            /* Height of message window */
#define Lmsg 500           /* Length of message window */
#define window_length 400  /* Window length */
#define window_height ((Ytop-Ybot)-(No_of_windows-1)*Yspace)/No_of_windows
#define max_Nwindows  8    /* maximum # of windows */
#define msg_win  8         /* Message window # */
#define max_win_height 75  /* maximum window height */
#define Xend (Xstart+window_length)
#define back_color 0       /* black */
#define msg_color 3        /* orange */
int No_of_windows=output_lay_length;          /* initial # of windows */
int winFlag=RESET;              /* flag (SET) indicating windows are open */
int Point1Flag[max_Nwindows]={0}; /* first plotting point flag (SET) */
double Yscale[max_Nwindows];      /* Y-coordinate scale factor */
double Yoffset[max_Nwindows];     /* Y-coordinate offset */
int Line_Color[max_Nwindows];

void open_windows()
{
int wi;       /* window index */
char *sp,buf[90];
double min_val,max_val;/*Window limits values*/
int     xll,yll,/*lower left coordinate*/
        xur,yur;/*upper right coordinate*/

        for(wi=0;wi<No_of_windows;wi++){
        yur=Ytop-wi*(window_height+Yspace);
        xur=Xstart+window_length+1;
        xll=Xstart;
        yll=yur-window_height;
        ug_window(wi+1,back_color,xll,yll,xur,yur);
        ug_box(wi+1,7,0,xll,yll,xur,yur,0);
        sprintf(buf,"\nWindow %i min_val [0] ",(wi+1));
        PutStr(buf);sp=GetStr();
                if(!*sp) sprintf(sp,"0");
        sscanf(sp,"%f",&min_val);
        ug_puts(wi+1,7,0,1,1,sp,0);
        sprintf(buf,"\nWindow %i max_val [1]  ",(wi+1));
        PutStr(buf);sp=GetStr();
                if(!*sp) sprintf(sp,"1");
        sscanf(sp,"%f",&max_val);
        ug_puts(wi+1,7,0,1,(window_height-15),sp,0);
        Yscale[wi]=window_height/(max_val-min_val);
        PutStr(buf);
        Yoffset[wi]=min_val;
        Line_Color[wi]=wi+1;
        }
winFlag=SET;
}

void PlotData(wi,Yval)     /* plot next data point */
double Yval;
int wi;
{
int y;       /*(relative) y-coordinate */
static int Xold[max_Nwindows]={0};          /*previous X plotting point*/
static int Yold[max_Nwindows];              /*previous Y plotting point*/

/* compute where the point should be plotted */
y=(Yval-Yoffset[wi])*Yscale[wi];
if(y<1) y=1;
if(y>=window_height) y=window_height-1;

/* clear old cursor out */
if(Xold[wi]) ug_line(wi+1,back_color,0,Xold[wi],1,Xold[wi],window_height-1,0);

/* plot new data point */
if(Xold[wi]==0 || Point1Flag[wi]==RESET)
ug_point(wi+1,Line_Color[wi],0,Xold[wi],y);
else ug_line(wi+1,Line_Color[wi],0,Xold[wi]-1,Yold[wi],Xold[wi],y,0);

/* save values for posterity */
Yold[wi]=y;
Point1Flag[wi]=SET; /* indicate that first point already plotted */

/* compute next curser location */
Xold[wi]++;
if(Xold[wi]>=window_length) Xold[wi]=0;

/* draw new curser line */
ug_line(wi+1,15,0,Xold[wi],1,Xold[wi],window_height-1,0);
}
```

```
csv3.1              !file format is Control Strategy Version 3.1
! Feb 92  (emulbpt.nnc)  Back Propagation Network
!**************************************************************************
!*                                                                       *
!*    Back Propagation Network Control Strategy, with inupt and output*
!*    Regression layers. Suitable for N models (N>=1).                   *
!*                                                                       *
!*    Modified by: Shahar Dror. February 26th 1992.                      *
!* 1.This control strategy allows to test in recall mode, while loading*
!*  the desired output via the 'read' function. In Learn mode it works*
!*  identically to the original backprop control strategy.              *
!*   Here is an example how to utilize it:                               *
!*    NINT UIO_Read(LayN, nPEs, Datap)                                   *
!*    NINT  LayN;                                                        *
!*    NINT  nPEs;                                                        *
!*    SREAL *Datap;                                                      *
!*    {                                                                  *
!*       if(LayN==Out_lay)  return(UIO_Learn_Output(LayN,nPEs,Datap));*
!*       else                return( UIO_Learn_Input(LayN,nPEs,Datap));  *
!*    }                                                                  *
!*                                                                       *
!*    Revision date:       2 March 92                                    *
!* 2. The control strategy supports a network with built in delay line*
!*  (regressor) form the output nodes and from the input nodes.          *
!*    In the learning phase, the desired output (in USRIO) and the       *
!*  outputs of the regressors are saved (io lrnrslt for layers 1&2).     *
!*  Before the next learning presentation of that same model, the values*
!*  are       fed into these layers as if they were there for normal BackProp*
!*  operation.                                                           *
!*  The values are loaded to the output field of the PE.                 *
!*    Revision date:       31 March 92                                   *
!* 3. The network has to have an output buffer layer, and the regress'n*
!*  are taken from the buffer layer (not from the output). This is due*
!*  to  NeuralWare implementation (feedback from the output layer always*
!*  takes the desired value and not the output value.                    *
!*  ....                                                                 *
!*    A similar control strategy is used for training controllers. For*
!*  uniformity, the AUX2 counter comparison was put so that the CS will *
!*  in cases where there is no need to extract information from a middle*
!*  layer. The USRIO program controls the value of AUX2 counter.         *
!* 4.  In testing the CS supplies "correct" values to the output regress*
!*  layer, to check if the learning was ok. In the real game there is a*
!*  direct feedback from the output (practically from the output buffer*
!*  layer), but since the training is not always 100% accurate and the*
!*  recurrent nature of the network, it causes the results to wonder.    *
!*                                                                       *
!*    Revision date:       8 April 92                                    *
!* 5.  In Recall/Test mode, when a new input is tested, the regressors*
!*  (OutReg & ComReg), that contain the regressed data of the previous*
!*  input that was under test, are initialized. The counter WRK1 is used*
!*  as a switch to determine when to initialize the regressors, and is*
!*  controlled from the USRIO program.                                   *
!**************************************************************************
! 07-Sep-90 Modified to work in learn mode with confusion matrices
!
!MASK          label  op-code        operands    comment
L_saR_sa       trace  aux3           !  0 set trace option to aux3
!L_saR_sa      optclr op:bknc        !  1 do not BKp to PEs w/o conns. BUG!sd
Li_aR_sa       cset   recall,0       !  2 recall count
!
! Get input (learn and recall) and desired output (learn only)
!
L_saR_sa       lset   in             !  3 input layer
L___R_sa       io     read           !  4 get input data (recall)
L_saR___       io     lrnin          !  5 get input data (learn)
L_saR___       lset   cur,1          ! command(input) regression layer(rev.2)
L_saR___       io     lrnin          ! get ComReg(model#) last output(!) data
L_saR___       lset   cur,1          ! output regression layer   (rev.2)
L_saR___       io     lrnin          ! get OutReg(model#) last output(!) data
!
! On the start of test of new command input, initialize ComReg & OutReg.
! The counter wrk1 serves as the switch and is controlled from the USRIO.
! (rev. 5)
L___R_sa       ccmp   wrk1,1         !is wrk1<1
L___R_sa     blt    @noreg          !do not initailize ComReg & OutReg
L___R_sa     lset   in,1            ! command(input) regression layer
L___R_sa     io     read            !initialize ComReg(model#,input#) (test)
L___R_sa     lset   cur,1           ! output regression layer
L___R_sa     io     read            !initialize OutReg(model#,input#) (test)
!
L_saR_sa @noreg     lset   out,-1   ! output buffer layer
L_saR___     io     lrnin           ! get output(model#)(!) data (learn)
L___R_sa     io     read            ! get output(model#) data(recall,rev. 4)
L_saR_sa     lset   out             !  6 output layer
L_saR___     io     lrnout          !  7 get desired output (learn)
L___R_sa     io     read            !  8 get desired output (test) (rev. 1)
!L___R_sa    io     rcltst          !  8 get desired output (test)
!
! Start with the first layer for a forward pass through network
!
L_saR_sa     lset   in              !  9 input layer
L___R_sa @rloop     math   sum¦rnoise¦tran¦output¦e=0 ! 10
L_saR___     math   sum¦lnoise¦tran¦output¦e=0¦fire ! 11
L_saR_sa     lset   cur,1           ! 12 next layer
L_saR_sa     lcmp   out             ! 13 at output layer ?
L_saR_sa     blt    @rloop          ! 14 loop till done
!
! Compute final outputs
!
L___R_sa     math    sum¦rnoise¦tran¦output¦e-=w  !15 Recall
L_saR___     math   sum¦lnoise¦tran¦output¦e-=w¦e*=ef¦fire ! 16 learn
!
! Write results to userio
!
L_saR___     lset   in,1            !set command(input) regressor layer(rev2
L_saR___     io     lrnrslt         !save output for next learn of that modl
L_saR___     lset   cur,1           !set output regressor layer (rev.2)
L_saR___     io     lrnrslt         !save output for next learn of that modl
L_saR_sa     ccmp   aux2,2          !is aux2<2 (rev. 3)
L_saR_sa     blt    @noeff          !go to output layer (no effector layer)
L_saR_sa     lset   in,aux2         !set effector layer
L_saR_sa     io     lrnrslt         !write its values (for control purposes)
L_saR_sa @noeff     lset   out      !set output layer
L___R_sa     math    ce=e           ! 17 Recall
L___R_sa     io     write           ! 18 recall
L_saR___     io     lrnrslt         ! 19 learn
!
! Learn cycle - back propagate error.  Store unscaled error in
! current error field.
!
L_saR___     lset   out             ! 20 set output layer
L_saR___ @lloop     math   ce=e¦e*=f'¦backp¦learn¦fire ! 21
L_saR___     lset   cur,-1          ! 22 previous layer
L_saR___     lcmp   in              ! 23 at input layer ?
L_saR___     bgt    @lloop          ! 24 loop till done
L_saR_sa     trace  0               ! 25 turn off any trace function
```

LIST OF REFERENCES

[AC89] Charles W. Anderson, "Learning to Control an Inverted Pendulum Using Neural Networks," *IEEE Control Systems Magazine*, April 1989 pp. 31-37.

[AD91] David Aviv, "Detection of Abrupt changes in Statistical Models," Doctoral dissertation, Naval Postgraduate School, Monterey, California, 1991.

[AF66] Michael Athans, and Peter L. Falb, *Optimal Control: An Introduction to the Theory and Its Applications*, McGraw-Hill, 1966.

[AW89] Karl J. Åström, and Björn Wittenmark, *Adaptive Control*, Addison-Wesley, 1989.

[BC92] Denis J.S.R. Bertrand, and Daniel J. Collins, "Neural Network Controller for the X29 Aircraft," *Proc. of the International Joint Conference on Neural Networks*, Vol.I, pp.321-326, Baltimore, Maryland, June 1992.

[BD91] Denis J.S.R. Bertrand, "Application of Neural Network to Adaptive Control Theory for Super-Augmented Aircraft," Engineer's Thesis, Naval Postgraduate School, Monterey, California, 1991.

[BH75] Arthur E. Bryson Jr., and Yu-Chi Ho, *Applied Optimal Control*, John Wiley and Sons, New York, 1975.

[BS90] George H. Burgin, and Steven S. Schnetzler, "Artificial Neural Network in Flight Control and Flight Management Systems," *IEEE National Aerospace and Electronics Conference*, Vol. 2, pp. 567-573, 1990.

[BSA83] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson, "Neurolike Adaptive Elements that Can Solve Difficult Learning Control Problems," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-13, No.5, September/October 1983.

[CF90] Fu-Chuang Chen, "Back-Propagation Neural Network for Nonlinear Self-Tuning Adaptive Control," *IEEE Control Systems Magazine*, April 1990 pp. 44-48.

[CS88] Richard Y. Chiang, and Michael G. Safonov, *Robust Control Toolbox, for Use with MATLAB™, User's Guide*, The MathWorks, Inc., 1988.

[DS89] Shahar Dror, "State Estimator for Maneuverable Target Using Piecewise Constant Acceleration Command," Master's Thesis, Technion - Israel Institute of Technology, Haifa, 1989 (in Hebrew).

[GEK88] Allon Guez, James L. Eilbert, and Moshe Kam, "Neural Network Architecture for Control," *IEEE Control Systems Magazine*, April 1988 pp. 22-25.

[GF90] William Goldenthal, and Jay Farrell, "Application of Neural Networks to Automatic Control," AIAA 90-3438-CP, 1990.

[GLLT90] Andrew Grace, Alan J. Laub, John N. Little, and Clay Thompson, *Control System Toolbox, for Use with MATLAB™, User's Guide*, The MathWorks, Inc., 1990.

[GM89] Michael Gherrity, "A Learning Algorithm for Analog, Fully Recurrent Neural Network," *Proc. of the International Joint Conference on Neural Networks*, Vol.I, pp.643-644, June 1989.

[HG90] Gerald A. Hartley, "F-18 Robust Control Design Using H2 and H-Infinity Methods," Master's Thesis, Naval Postgraduate School, Monterey, California, 1990.

[HN91] Robert Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, 1991.

[HSW89] Kurt Hornik, Maxwell Stinchcombe and Halbert White, "Multi Layer Feedforward Networks are Universal Approximators," *Neural Networks*, Vol.2, pp. 359-366, March 1989.

[HWB92] C.M. Ha, Y.P. Wei, and J.A.Bessolo, "Identification of Aircraft Dynamical Systems via Neural Networks," AIAA 92-0047, 1992.

[HWB92] C.M. Ha, Y.P. Wei, and J.A.Bessolo, "Reconfigurable Aircraft Flight Control System via Neural Networks," AIAA 92-1075, 1992.

[IA89] Albert Isidori, *Nonlinear Control Systems: An Introduction*, 2nd Edition, Springer-Verlag, Berlin, Germany, 1989.

[JI89] H.I. Jordan, "Generic Constrains on Underspecified Target Trajectories," *Proc. of International Joint Conference on Neural Networks*, Vol. I, pp. 217-255, Washington, 1989.

[KC90] L. Gordon Kraft, and David P. Campagna, "A Comparison Between CMAC Neural Network Control and Two Traditional Adaptive Control Systems," *IEEE Control Systems Magazine*, April 1990 pp. 36-43.

[KIS88] Mitsuo Kawato, Yoji Uno, and Ryoji Suzuki, "Hierarchical Neural Network Model for Voluntary Movement with Application to Robotics," *IEEE Control Systems Magazine*, April 1988 pp. 8-15.

[KS72] Huilbert Kwakernaak, and Raphael Sivan, *Linear Optimal Control Systems*, Wiley-Interscience, New-York,1972.

[KT80] Thomas Kailath, *Linear Systems*, Prantice-Hall, Inc., 1980.

[LGI91] Esther Levin, Raanan Gewirtzman, and Gideon F. Inbar, "Neural Networks Architecture for Adaptive System Modeling and Control," *Neural Networks*, Vol. 4, pp. 185-191, 1991.

[LL87] Lennart Ljung, *System Identification: Theory for the User*, Prantice-Hall, Inc. New-Jersy, 1987.

[MAG90] Duane McRuer, Irving Ashkenas, and Dunstan Graham, *Aircraft Dynamics and Automatic Control*, Princeton, New Jersey,1990.

[MDC82] McDonnell Douglas Corporation Report No. MDC A7813, *F/A-18A Flight Control System Design Report Vol. I and II, System Description and Theory of Operation,* December 1982.

[MS80] Military Specification, Flying Qualities of Piloted Airplanes, MIL-F-8785C, 5 November 1980.

[MSW90] W. Thomas Miller,III, Richard S. Sutton, and Paul J. Werbos eds., *Neural Networks for Control*, The MIT Press, 1990.

[NC91] *Neural Computing*, NeuralWare, Inc., 1991.

[NP90] Kumpati S. Narendra, and Kannan Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," *IEEE Tran. Neural Networks*, Vol. 1, No. 1, pp. 4-27, March 1990.

[NP91] Kumpati S. Narendra, and Kannan Parthasarathy, "Gradient Methods for Optimizing of Dynamical Systems Containing Neural Networks," *IEEE Tran. Neural Networks*, Vol. 2, No. 2, pp. 252-262, March 1991.

[NW90] Derrick H. Nguyen, and Bernard Widrow, "Neural Networks for Self-Learning Control Systems," *IEEE Control Systems Magazine*, April 1990 pp. 18-23.

[PSY88] Demitri Psaltis, Athanasios Sidris, and Alan A. Yamamura, "A Multilayered Neural Network Controller," *IEEE Control Systems Magazine*, April 1988 pp. 17-21.

[RF86] Fredric W. Rojek, "Development of a Mathematical Model That Simulates the Longitudinal, and Lateral-Directional Response of the F/A-18 for the Study of Flight Control Reconfiguration," Master's Thesis, Naval Postgraduate School, Monterey, California, 1986.

[RF91] *Reference Guide for NeuralWorks Professional II/plus and NeuralWorks Explorer*, NeuralWare, Inc. 1991.

[RM86] D.E. Rumelhart, J.L. McClelland et al, *Parallel Distributed Processing*, The MIT press, 1986.

[SC90] Russell W. Scott, and Daniel J. Collins, "Neural Networks Adaptive Controllers," *Proc. of the International Joint Conference on Neural Networks*, San Diego, California, June 1990.

[SD91] Donald F. Specht, "A General Regression Neural Network," *IEEE Tran. Neural Networks*, Vol. 2, No. 6, pp. 568-576, November 1991.

[SL91] David R. Seidi, and Robert D. Lorenz, "A Structure by which a Recurrent Neural Network Can Approximate a Nonlinear Dynamic System," *Proc. of International Joint Conference On Neural Networks*, Vol. II, pp.709-714, July 1991.

[SM92] M. Steinberg, "Potential Role of Neural Networks and Fuzzy Logic in Flight Control Design and Development," AIAA 92-0999, 1992.

[SR89] Russell W. Scott II, "Application of Neural Network to Adaptive Control ," Engineer's Thesis, Naval Postgraduate School, Monterey, California, 1989

[SS89] Marco Saerens and Alain Soquet, "A Neural Controller Based on Back-propagation algorithm," *Proc. of the First IEE International Conference on Artificial Neural Networks*, pp.211-215, London, 1989.

[SSRB90] Marco Saerens, Alain Soquet, Jean-Michel Renders, and Hugues Bersini, "Preliminary Comparisons Between a Neural Adaptive Controller and a

Model Reference Adaptive Controller," *Proc. of the International Neural Network Conference*, Paris, 1990.

[TC91] Chin Hock Teo, "Back-Propagation Neural Network in Adaptive Control of Unknown Nonlinear Systems," Master's Thesis, Naval Postgraduate School, Monterey, California, 1991.

[WP90] Paul J. Werbos, "Consistency of HDP Applied to a Simple Reinforcement Learning Problem," *Neural Networks*, Vol. 3, pp. 179-189, 1990.

INITIAL DISTRIBUTION LIST

1.	Defence Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Chairman, Code AA Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5000	1
4.	D.J. Collins, Code AA/Co Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5000	2
5.	B.N. Agrawal, Code AA/Ag Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5000	1
6.	L.V. Schmidt, Code AA/Sc Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5000	1
7.	A. Schoenstadt, Code MA/Zh Department of Mathematics Naval Postgraduate School Monterey, California 93943-5000	1
8.	H. Titus, Code EC/Ts Department of Electrical Engineering Naval Postgraduate School Monterey, California 93943-5000	1

9.	R. Kolar, Code AA/Kj Department of Aeronautics and Astronautics Naval Postgraduate School Monterey, California 93943-5000	1
10.	Naval Attache Israeli Embassy 3514 International Drive, N.W. Washington D.C. 20008	1
11.	LCDR. Shahar Dror 41 Vitkin St. Nathanya, 42376 ISRAEL	2